A Simple Introduction to Game Programming With C# and XNA 3.1

No Previous Programming Experience Required

Curtis Bennett
xnagamemaking.com

A Simple Introduction to Game Programming With C# and XNA 3.1 By Curtis Bennett

Printed in the United States of America.

First Edition: 2009

This book is an updated edition of *Creating 2D Games With XNA: A Simple Introduction to Game Programming With C# and XNA* from 2008.

For sample code go to: xnagamemaking.com

The author's website: curtisbennett.com

The free ebook edition of this book is sponsored by:

Makers of the XNA game:

Look for it on the Xbox LIVE Indie Games Channel under the category Racing and Flying and the title:

Charlie Cat's Hot Air Balloon.

Table of Contents

Introduction

Microsoft's XNA Game Studio is becoming the most popular platform for students and beginning developers to learn game programming. This book, *A Simple Introduction to Game Programming With C# and XNA 3.1: No Previous Programming Experience Required,* aims to give people with a limited exposure to game programming an introduction to get them started. This is by no means a comprehensive introduction, but just goes over the basics.

Use In Education

Material from this book has been used at the community college and high school level, and I've received reports that younger kids are using it too to learn the basics of programming.

Why Use This Book to Learn XNA?

There are several popular XNA books, so why bother reading this one? When trying to find an introductory XNA book for the college course I found the options limited. Not that the books were bad, they just didn't suit the purpose. Some were too advanced for beginners, others focused on specialized issues (such as making helper libraries or spending significant time on optimization) and weren't general enough for a class. What I needed was a complete basic introduction to C# that XNA aimed at beginners.

That is what makes *A Simple Introduction to Game Programming With C# and XNA 3.1*different from other books: it is aimed at complete beginners, not only people with no programming experience but those without much technical experience. This book also focuses on basic game programming concepts more than specific syntax. It is more of a basic programming book using XNA than a book that just lists how to do simple tasks in XNA. The book also keeps things simple by only covering 2D games and would be ideal for a high school or lower level college game programming course.

The material in *A Simple Introduction to Game Programming With C# and XNA 3.1* has also been tested on students and has been adjusted based on feedback For instance, I was surprised to find that many students had difficulty in abstracting a basic player class. Putting in a sprite and moving it around on screen made sense, but they had trouble encapsulating this into a separate class. This issue became an entire chapter in the book (Chapter 7 – Creating a Player Class) to help students get over this hurdle. And I found that audio was not a major issue, and people can learn how to use it from the documentation and so it is not mentioned here.

Target Audience

The main audience for this book is people interested in learning game programming but who have no previous experience in programming and who know little about making games. This book is also for developers who may have some programming experience but limited exposure to game programming (such as looking at tutorials online.)

This book is not for experienced game developers who are wanting to learn XNA, or programmers who are looking for how to implement advanced effects with XNA, as they will find the scope of the book too limited.

Contents

Part I - An Introduction to Programming and C#

In this part of the book a basic introduction is presented on C#. An emphasis is placed on the basic concepts of programming and object-oriented design. This section has enough C# to get people started making 2D games in the next section.

Part II - Making 2D Games in XNA

The second part of this book presents a thorough introduction to 2D game programming. All of the major components, parallax backgrounds, animation, enemy interactions, adding start screens and particle systems are covered. We also walk through creating a complete game, a 2D spaceship scroller.

Also note that all of the source code and samples are up at xnagamemaking.com.

Feedback

Thank you for taking the time to look through this book. Any feedback (or corrections) is appreciated. I can be reached at curtis@xnagamemaking.com.

Curtis Bennett

Part I: An Introduction to Programming and C#

Game programming is a lot of fun. Whether you want to learn game programming as a hobby or to become a professional developer, programming games is very rewarding. But a lot of people, when they first decide to learn game programming, want to start making full 3D games the first day. When I teach game programming I'll get questions like “How do I make a game like Halo 3?” or “How do I program a MMORPG?” in the first class. Some people are disappointed to hear that we can't jump into doing those kinds of things right away. Before making a 3D game we need to learn the basics of game programming and the building blocks of all programming. That's what we'll do in this first part of the book; we'll go over the basics of programming and C#. We won't even get to graphics yet; all of our programs will be Console programs, which are text based. The example programs here are all little text ones and we'll even develop a very simple text-based “game” in the second through third chapters and some basic ship game code in chapters four and five. (I put game in quotes because it will really be just the start of a game; we'll save the larger and complete games for XNA) We'll be going through this material as quickly as possible, so we can get to the more exciting Part II where we start making our 2D games.

Chapter 1: An Introduction to Programming

In this chapter we'll start programming. We'll start at the very beginning, discussing the very basics of what programming is and the different paradigms of programming languages. Then we'll make our first C# program. Specifically we'll do the following:

- Learn the basic concepts of programming
- Look at the different types of programming languages available
- Setup Visual C# Express and XNA 3.1
- Create a simple C# program

Programming Basics

So what exactly is a computer program? We can say that a computer program is just a list of instructions written by someone (a programmer) that can be given to a computer. The instructions (called computer code or just code) are written in a programming language that the computer can understand, and when the instructions run on the computer we say the program is being executed.

You may have heard before that computers understand nothing except ones and zeros and that's true. The language of computers is binary, which is just long lines of zeros and ones. Everything you see, all the cool artwork in the latest games, comes down to strings of binary. In the very early days of computers people worked with binary directly, trying to input representations of ones and zeros into the computer somehow. But binary, while easy for computers to understand, is difficult for people, so computer scientists started making special letter combinations to represent different computer commands from the binary. For instance, let's say the programmer wanted to tell the computer to add two numbers. The numbers are stored in memory at places called R1 and R2

and the programmer wants to put their sum at place R3, the programmer would input something like this:

```
ADD R1, R2, R3
```

This is called assembly language and for years computer programs were written in this, and all of the video games were programmed with it (and it is still used in graphics today in shaders.) But writing computer programs with assembly is still not very straightforward. Seeing one or a few lines of assembly code is simple enough but imagine trying to deal with thousands of lines of it. This lead people to create high level programming languages like C++, Java, or C#. These languages let people program computers in a way that is easier for the programmer to understand. For example, the previous line of assembly code would be written like this in C#:

```
r1 = r2 + r3;
```

Except that instead of r's for the names of the numbers we'd have something more meaningful, like:

```
totalGold = goldInBag+ goldInTreasureChest;
```

This is a statement, a line of computer code. The files that contain code are called source files. With a high level language we need a way to convert these high level statements into instructions that the computer can understand. To do this we use a program called a compiler. A compiler takes in source files and changes the instructions to a form the computer can understand (which isn't binary directly; it changes to other code for the operating system, but we won't worry about those details.) The compiler creates an executable program that can be run on the computer. This process of taking the source file(s) and creating the executable is called compiling or building the program. The compiler we are going to use is Visual C# Express, a very popular and powerful compiler from Microsoft, and we'll go over setting it up later in this chapter.

As for the programming language we'll use C# (pronounced C sharp.) It is our language of choice for, among other reasons, it's the only language that works with XNA. C# is only one of many other programming languages (some of the most popular are C++, Java, and Visual Basic.) They each have their own strengths and weaknesses. C# is similar to Java and the language C++ (and C++ is an extended version of the C language.) Some beginners, when hearing that C# was partially based off of C++, think it's a good idea to first try to learn C++ or C before C#. But that is really unnecessary, and can make things more confusing trying to learn two or three languages instead of one. C# is actually a lot easier to use than C++ (at least in my opinion) and there's no advantage to learning C++ if you're working with C#. (But note that currently C++ is the most popular language to program games. While C# may become the dominant language, today all of the larger and powerful games use C++, so it would be good to learn C++ after C#.)

We should also note that C# uses the .Net framework; a huge collection of classes that provide functionality for a wide range of applications. We'll also be using XNA 3.1. XNA is a special add-on that has game programming functionality which obviously we'll learn a lot about later. But before getting into much info on C# specifically, let's look a bit at programming languages in general.

Types of Languages

Not all programming languages are created equal, and it's good to know about the different types of languages available. While all computer code eventually turns into binary for the computer, before that they have a lot of differences. The most obvious difference between various programming languages is their syntax, the rules for how they are written. Syntax is an important issue for programming, as all computer instructions must be typed in a very strict way, the littlest change from the way a computer statement is supposed to be will give an error. For instance, part of C#'s syntax is that at the end of each instruction there is a semicolon (like at the end of the adding example before) and without the semicolon an error will result. This semicolon rule is pretty common, but some languages like Visual Basic don't use a semicolon for that. But

besides syntax differences computer languages can also differ in the way they are designed; their whole way of being organized and structured. There are only a few major paradigms (types) of programming languages. Let's look at a few.

Structured (Procedural) Languages

Structured languages are written in a very straightforward way. They start at the beginning and just list all of the computer instructions one right after the other. A program is made up of a (usually) very long sequential list of commands telling the computer what to do. The computer executes (does) the instructions in order, doing statement one first, then statement two, and so on. A structured program looks something like this:

```
instruction 1;
instruction 2;
...
instruction 5000;
...
```

If you've ever taken a course in school or tried programming in BASIC, then you've used a structured language. This was the original and major paradigm for computer languages for years, and is still popular today. The most popular language of this type is called C (and C#'s syntax is based on it.) Looking at our program above again, you can imagine that as these types of programs become very long you'll need to repeat some of the code. Let's say in several parts in a program we need to convert a temperature from Celsius to Fahrenheit and that code will be repeated over and over. To handle this structured languages are usually block based, meaning instead of one long list of code they have functions (blocks) of code that each perform separate tasks. The program for the temperature could look like the following:

```
function changeTemperature
(some instructions here)
function end
```

```
instruction 1;
changeTemperature;
instruction 2;
...
```

The temperature conversion function is defined in the beginning of the program, and after instruction 1 we call the function changeTemperature and do all of the code inside of it. Then we continue with other code and will call it again later when needed. This has many big advantages. The main one being we don't have to repeat typing in computer code; we can write something once and then call it as a function when we need to. Using functions also helps organize the code; we can put separate functions together in separate source files. Functions also help with a host of other issues we'll look at later.

But structured programming does have disadvantages. For starters, having all of these functions is fine up to a point but as programs become very large they can start to be confusing, and errors in the code can be hard to find. But more importantly, structured programs are written in a way for computers to understand, not for humans to understand. If we started designing, say, a car game we would start describing it in ways like see a car on screen, press up to accelerate, down to break, have to hit this checkpoint before time runs out, etc. When we go to write a structured program to do this it's not always easy to change that design to a linear list of code. The game isn't thought of as just a series of steps, and it'd be nice if our programs could be coded in a way that reflected that. So developers came up with our next type of programming language:

Object-Oriented Languages

Instead of looking at the car game and asking what series of steps we can do to describe a car in a computer program, it would be nice if we had a programming language that could describe a car in more human terms. Instead of thinking of steps to program a car we would like to program a car game by creating a car object and giving it a description. Object-Oriented programming (OO) does just that. Instead of thinking of a series of instructions for a car, we could first think how we can describe a

car, what model is it, what color, or describing how fast it's going. We can then think of what kind of functionality a car has, such as we can accelerate a car, turn it, or brake, etc. We can treat the car as an object, and OO programming allows us to think of the real world and program in terms of objects. We make classes of objects, which are descriptions for types of objects. Classes are designs for objects, a template for what they are like. The actual objects themselves are called instances of a class. For example, we could make a very specific car class to describe Honda Accord cars, the class would contain information all Accords have (such as year, mileage, color, etc) But this class would only described the car in general, an actual Honda Accord sitting in a parking lot outside of the building I'm in right now would be an instance of the Honda Accord class. The car in the parking lot is not a class of cars; it is a specific instance of a car. This difference between classes/instances may not seem that important now but will become an important issue later.

Anyways, let's look at making just a general simple car class. We can create a class like the following:

```
class Car
{
    modelType;
    currentSpeed;

    accelerate();
    turnLeft();
    turnRight();
    brake();
}
```

Classes are made out of a combination of data (descriptive properties like the modelType and currentSpeed) and functions (actions that do something to the object like the accelerate() or turn left(). We can tell what is a function because they end in the two parentheses, (). Some people think of objects as nothing more than just data and functions, only a good way to organize programs. Others say that the objects in programs should be very robust and mimic real life, I've heard OO proponents say they're angry over how the term "window" is used in computers, as a window is something transparent you see through and

that doesn't match up with its computer counterpart. Object-Oriented Design, choosing what the different objects and classes are going to be for a program, is huge field. In our programs and games, though; we'll just try to keep them as straightforward as possible.

Overall OO programming is the main paradigm in programming today. All of the most popular languages, C#, C++, Java are Object-Oriented, and even languages like Visual Basic which used to not be OO are becoming that way. There are just so many advantages to this way of programming. Besides it being easier to understand and organize programs, OO programming lends itself to be used by teams since different team members can easily do different objects for a system. We've just scratched the surface of OO programming here, but we'll look at it in more detail later in the book.

Lists/Logic Languages

Structured Programming and OO programming are the two major paradigms of programming, all of the very popular languages fall into one of them. But this doesn't mean that all programming languages work that way. Some languages work in very unique and original ways. For instance, the language SML uses lists as its way of programming. You start with a list (a string of characters) and this list is manipulated by the program as it goes through it ending with a different list at the end (if this sounds strange or confusing don't worry about it, as most programmers have a hard time understanding languages that aren't structured or OO) Other languages use logic to make programs, inputting in logical rules from philosophy to create programs. These other paradigm languages are usually very good at doing a few things (For instance, SML can do complex list manipulations much easier than C# or C++) but aren't general enough to do general programming tasks. And outside of making a few tools for making games, aren't used in the industry.

Type of C#

So we've been through the major types of programming languages, but what is C#? Well earlier we did say that C# is Object-Oriented and it is, everything in C# is an object, and all the code is about manipulating

objects. It uses the .Net framework, which is a huge collection of classes (objects) to use. Likewise XNA is a collection of classes to use too, which are made especially for game programming. But to be more specific, C# is called a block-structured Object-Oriented language, which means that while objects are used for everything, the functions (or methods to be more precise, methods are a special type of functions), are contained in blocks, which seem like little procedural programs. In fact, the programs we'll be doing in the first part of the book will seem very similar to procedural programs, until we get to the part on OO.

Our First Program

But enough theory, let's get down to making our first program.

Setting up C# and XNA

Before we can program anything we'll need to set up our programming environment, that is to say Visual C# Express and XNA. I could go through the steps to setting up XNA in detail, but Microsoft has a lot of good documentation set up for this; we'll just mention the main points. For details go to: creators.xna.com/en-US/quickstart_main. Also note if you have a full commercial version of Visual Studio you can use that instead of the Express edition.

To setup XNA download and install the following:

- Visual C# 2008 Express Edition
- XNA Game Studio 3.1

Installing these is simple, just go to creators.xna.com/en-US/downloads and click on the appropriate links. Again go to creators.xna.com/en-US/quickstart_main for more details.

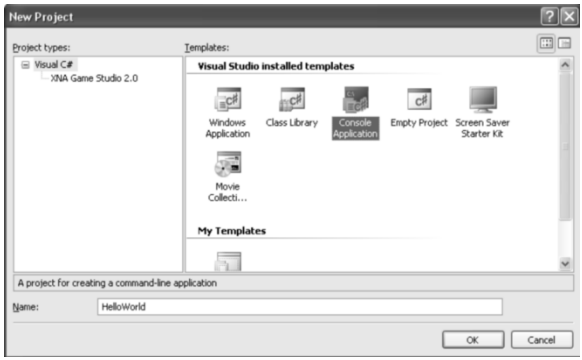
First C# Program

Now that we have Visual C# Express setup let's go straight to making our first program. Go ahead and start Visual C# by going to Start->All

Programs->Microsoft Visual C# Express 2008. You'll see a screen similar to the figure below.

Our first program is the traditional first program of any programming language; we'll make a program that prints "Hello World!" on the screen. That may not sound like the most exciting thing out there, but it's an accomplishment to create a program and execute it.

Our first step is to create a project. All code in C# is organized into projects, which organize source files and resources. In Visual C# Express Go to File->New Project. There are several program types, but select the icon Console Application, and in the name type "HelloWorld" (without the quotation marks).



Then click OK. The code below is then displayed in the left side of the window:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace HelloWorld
{
   class Program
   {
      static void Main(string[] args)
      {
      }
   }
}
```

We'll go through this line by line in a moment, but for now just type in the following line in between the two brackets {} after the string void Main line:

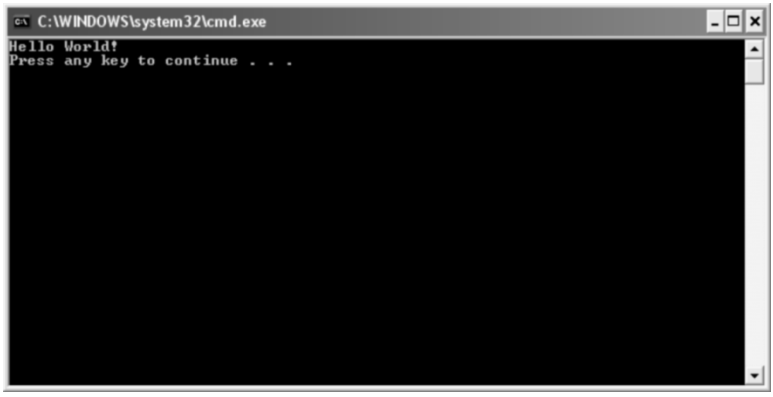
```
Console.WriteLine("Hello World!");
```

The complete program should look like this:

```
using System;
using System.Collections.Generic;
```

```
using System.Linq;
using System.Text;

namespace HelloWorld
{
   class Program
   {
      static void Main(string[] args)
      {
         Console.WriteLine("Hello World!");
      }
   }
}
```

Next go to Debug->Start Without Debugging (or hit Ctrl+F5). The following will display on the screen:



Simple as that we have our first program running. If you have any problems make sure the code is typed in exactly as in the listing. Let's go through it line by line, starting with:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
```

C# gives a lot of different instructions we can use in the program, but we want to tell C# which of the instructions we'll be using. (We don't want C# to give us every available instruction, since we won't use most of them and it adds overhead to the program to say we'll use more instructions then we need.) The first part of the program with the "using" statements tells C# what instructions we'll use and we'll leave this the same for our Console programs, but include XNA references when we get to part two.

The"using" instructions also let us type things in more compactly. In the Console instruction we put in above we could have written:

```
System.Console.WriteLine("Hello World!");
```

The using System line allows us to write the Console command without the System.

Note again that each of the commands ends with a semicolon; it is a way of telling C# that an instruction is complete. The amount of whitespace (spaces, tabs, returns) after an instruction doesn't matter, that the semicolon is there is all that counts.

Another note is that C# defines blocks of code by brackets {}. Looking at our code again notice everything is in three blocks. The first block, namespace HelloWorld, defines a namespace, which is just a general way to group code. The second block class Program defines an object, like what we talked about briefly in the Object-Oriented programming section before and we'll discuss objects more later in the book. The final block "static void Main(string[] args)" defines a function (or method) called Main, and is the starting point for the program. When we start the program the compiler starts running instructions from Main (and no matter how large a program becomes it only has one Main.)

As for the line of the program that does the actual work:

```
Console.WriteLine(“Hello World!”);
```

Is a pretty straightforward instruction to the Console object (the box that holds the text) telling it to write the line “Hello World!” on the screen. Statements like these are the working part of programs, and they are contained in object's functions (methods). And that's the whole program.

Comments

Besides regular code instructions we can add comments to programs. Comments are lines of text that are helpful notes to the programmer but are completely ignored by the compiler. C# has support for three types of comments. The first is a single line comment, helpful for little descriptions. It is merely two slashes, //, and everything after them is ignored:

```
// Print out Hello World!
Console.WriteLine("Hello World!");
```

The second type of comment is a multi line comment; these are helpful for giving longer descriptions of things. They are used by /* for the beginning and */ for the end, everything in-between being commented:

```
/*
 Print out Hello World!
*/
Console.WriteLine("Hello World!");
```

The third type of comment is for documenting code with XML, which is good, but we won't bother with it here for our programs.

Summary

In this chapter we had an introduction to computer programming and the different paradigms of programming languages out there. Then we setup Visual C# Express and created our first program. In the next chapter we'll look at programming more by seeing how programs handle data.

Chapter 2: Data and Operators

Data, that is to say information, is the main thing that programs work on. We can think of computer programs as doing nothing more than reading in data, doing some kind of manipulation on the data, and spitting it back out. This is true for all video game programs even if it may not seem immediately obvious. All games do is read in some art and other data, manipulate it to display it onscreen, then read in more data (keyboard presses, other input) and do some more things and output some more data (pictures, audio, etc.)

In this chapter we'll look at how data is used in programs and different operations that change it. Specifically we'll cover:

- What variables are and how to implement them in code.
- The different simple types of variables in C#
- C#'s basic operators
- Start making a simple text based game, "Finding the Gold"

Data

The simplest type of data is a simple variable. A variable is just a place in memory to store some information, and a name to describe where it is at. The information that is stored in the variable is called the value of the variable. Variables also have specific types (number, letter, string) depending on what kind of data they're storing. For instance the following:

```
int age;
```

Creates a variable of the integer type (a number) named age. We can then assign the variable a value, such as:

```
age = 142;
```

So later in the program (unless we change it) when we read what age is it will be 142. This is the basic concept of variables; let's go through some more facts about them.

Declaring Variables

Before we can use any variables we have to declare them; that is to tell the compiler about the variable before doing anything with it. The basic syntax of this is:

```
type name;
```

As in the age example above, we had to give the type, int for integer, and the name of the variable (age). The most common basic types are bool (true/false), int (integer), float (a decimal number) and char (a single character.) (We'll go over these types in more detail in a moment, but there are a few more points to cover first.) When we declare a variable we can also initialize it to a value. In the age example, we could have written it as:

```
int age = 142;
```

Variables should be assigned a value before they are used in C#.

Assigning Variables

We used the equal's symbol above to initialize the age variable to a value. Notice the equals sign in C# doesn't mean equals in the mathematical sense. It is actually the assignment operator; it takes the value to the right of it and assigns it to the variable on the left of the sign. Again for example

```
age = 142;
```

Says to take the value to the right of the equals sign, 142, and assign it to the variable age. This is called assigning the variable. (The equals sign in C# isn't =, but as we'll see in the next section, is two equals signs, ==.)

As we said before C# requires that a variable be initialized before it is used. A program like the following:

```
int test;
Console.WriteLine(test.ToString());
```

will generate an error for using the variable test when it is unassigned. In regular code not using an unassigned variable is a must and when we create objects it is not required (a default value will be created) but is still the preferred thing to do.

Simple Variable Types

Now that we looked at how variables are declared and how to store information in them (assign them values), let's go back over some of the different simple variable types we can use.

bool – boolean. Boolean variables are pretty simple, they either have one of two values, "true" or "false" These types come up when we start doing logic testing. There are a lot of places in programming where we need to decide if something is true or not (like if a game is currently paused) and these variables come in handy for that.

```
// A bool variable
bool testBool = true;
```

int – integer In case you don't remember from math class, integers are numbers that are "whole", in that they don't have any fractional components, no decimals. These numbers can be negative, positive, or zero. Integers come up a lot in programming; they are especially useful whenever we need to count something.

```
// An int variable
int testInt = 5;
```

float – floating point number These are decimal numbers, like 3.14, 0.0001, etc. Basically whenever we need any type of decimal number this is the type of variable we'll be using. The only catch with this is that any floating point number we type in we'll need to put a little "f" after it to say that it is a float. For example:

```
// float variables
float piGood = 3.14f; // Better, the little f ends our problems
float piBad = 3.14; // Compiler will call this an error
```

double – another decimal number. Doubles are similar to floats; they are also decimal numbers, only you don't have to include the little "f" after them. The difference between float and double is that double uses much more memory (we can think of it as using double the memory of a float.) We mostly use float for decimal numbers, as memory is important and we don't want to use excess if we don't have to, and (more importantly) the larger numbers are less efficient and take more time to do calculations than the floats. If on the other hand we are doing something very precise, like a complex physics calculation where we want to have as little error as possible, we'll need the numbers to be more precise (use more memory) and we'll use double instead of float.

```
// A double variable
double testDouble = 3.14;
```

char – a single character. Enough numbers, this simple type is a variable that just stores a single character as its value. The letter has to be put in single quotes.

```
// A char variable
char letter = 'a';
```

string –A series of characters. Not technically a simple type, but used just as much as the others. A string is a series of characters, like a line of text. The string, unlike the char, is denoted by putting it in regular quotes:

```
// A string variable
string lineOfText = “This is a test string”;
```

Strings come up everywhere in programming, and it is a class that we'll be using a lot. We'll cover strings in more detail in chapter 3.

These aren't all of the simple types we can use in C#. Each C# type also has an equivalent type in the .Net framework. If you're curious about the .Net types check the documentation in Visual C#. We'll only be using the simple types described above.

Type Conversion

We have all of these different variable types in C#, but the language is pretty strict about type conversion, converting from one type to another. The following code will generate an error:

```
float piFloat = 3.14f;
int piInt = piFloat;
```

This code will generate the error message:

```
Cannot implicitly convert type 'float' to 'int'. An explicit conversion exists (are you missing a cast?)
```

In some languages, like C++, assigning a float variable to an integer is OK (the pi code will compile in C++); the int variable will just ignore the decimal part. But in C# we need to make our conversions explicit, meaning we have to specifically state a conversion is going on. We do

this by putting the variable type we want to convert to in parenthesis is front of it:

```
float piFloat = 3.14f;
int piInt = (int) piFloat;
```

The above code will compile just fine (piInt will have a value of 3). We can also convert using the .Net Convert method, which we'll see a bit later in this chapter.

C# also does implicit conversion if the type being converted from is a subset of the type converting too. This means that since all ints are simpler than floats, we don't have to implicitly convert ints to floats.

```
// The following will run OK:
int basicNumber = 5;
float basicFloat = basicNumber;
```

Variable Naming

When we start making large games there will be many, many variables in them. One of the most important things to do when naming a variable is to make it clear and easy to recognize what the variable means. For example, let's say we're making a car game and we need to keep track of the maximum speed the car can go. A good variable name would be something like:

```
float maxSpeed;
```

But even this might not good enough if we're using different units for speed in different parts of the game (like miles per hour, feet per second.) A better name would then be:

```
float maxSpeedMph;
```

A bad variable name for maximum speed would be:

```
float ms;
```

This might be easier to type and at first ms for maximum speed makes sense. But later as the program grows the variable ms could be ambiguous (someone could confuse it for meters per second.) Always make variable names clear.

In addition we usually name simple variables using lowerCamelCase, which means multiple words are combined using a capital letter for each new word (and the first word is lower case) For class and functions names we use UpperCamelCase, which is like lower camel case except the first letter is also capitalized.

Mathematical Operators

So we now have that ability to declare variables and assign them values. let's look at how to do more things with them using operators. An operator is special function that does some action on a variable. The assignment (equals sign) is an operator; it performs the action of putting a new value in a variable. The first class of operators we'll take a quick look at our C#'s mathematical operators, these are operators we can use on our number variables, int, float, and double. The simplest are the addition and subtraction operators. They are pretty straightforward, we just use the plus (+) and minus symbols (-). Here's an example that shows how they work:

```
float startSpeed = 5.0f;
float currentSpeed= startSpeed + 5.0f;
// currentSpeed has 10.0f in it.
currentSpeed = startSpeed –  5.0f;
// currentSpeed has 5.0f in it
```

The multiplication and division operators work similarly, and for multiplication we use the star symbol * and for division the backslash /.

```
float startSpeed = 6.0f;
float currentSpeed = startSpeed * 3.0f;
// currentSpeed has 18.0f in it.
currentSpeed = startSpeed / 2.0f;
// currentSpeed has 3.0f in it
```

One possibly surprising thing is that when doing division with integers the remainder is ignored (since integers don't store a decimal part, they can't store remainders):

```
int testInt = 17 / 5;
// testInt now equals three, the remainder two is ignored.
```

Another mathematical operator is the modulus operator, which is basically the remainder operator. It performs division on two numbers and then outputs the remainder of them. The symbol for modulus is %: Here is an example:

```
int testInt = 17 % 5;
// testInt now equals 2, the remainder of 17 / 5
```

Unary Operators

Notice that whenever we do an operation like adding where we add two numbers and store the result in a third:

```
int testInt = testInt1 + testInt2;
```

the two numbers that were added didn't change; testInt1 and testInt2 have the same value after that line of code as they did before we entered it. Sometimes we need to do this. But sometimes we want to do something like this:

```
testInt = testInt + 5;
```

Where we add 5 to testInt and testInt itself is changed. This happens so often that there are special unary mathematical operators just for this occasion (unary meaning one) They work by putting the math symbol before the equals sign. The following two lines of code are the same:

```
testInt = testInt +5;
testInt += 5; // Same as the above

// This works the same for multiply/divide/subtract

testInt -= 5; // Same as testInt = testInt – 5;
testInt *= 5; // Same as testInt = testInt * 5;
testInt /= 5; // Same as testInt = testInt / 5;
```

Another very common thing to do that is even simpler is to add or subtract one from a number. This comes up so much that there are special operators for it. We put ++ to increment, add one to a number, and –- to decrement, to subtract one from a number. These can be put in front of or behind the number (there are some slight technical difference for putting in front of or behind a number but we won't worry about them here) Here's an example:

```
testInt = 5;
testInt--;
// testInt now equals 4
testInt++;
// added one to testInt, now it equals 5 again
```

Example Program

The following program is pretty straightforward; it just a quick example showing a few variables at work. The only new things are the Readline and Convert methods. The Console.Writeline method writes a string of text to the console, and Console.Readline() works the opposite way, pulling a string of text in from the console. When we have a string of text, such as in the example program when the user enters their age, we sometimes want to change the string into a different variable, like changing the age string to an int. To do this we use the Convert method.

Typing in Convert period will bring up a list of different types to convert a string to. The instruction:

```
int age = Convert.ToInt32(Console.ReadLine());
```

Reads in a string from the console and converts it to an int (and stores it in the variable age.) The dangerous thing with this code is if the user enters a string that won't convert to an int. This will throw an exception, that is, cause an error (which we won't worry about for now.) Here is the program:

```
static void Main (string[] args)
{

    int feet;
    int inches;

    Console.WriteLine("I'm going to ask about you height, feet and inches.");

    Console.WriteLine("How many feet tall are you?");
    feet = Convert.ToInt32(Console.ReadLine());
    Console.WriteLine("And how many inches?");
    inches = Convert.ToInt32(Console.ReadLine());

    float cm = ((feet * 12) + inches) * 2.54f;

    Console.WriteLine("You are " + cm.ToString() + " centimeters tall.");

}
```

A sample output is:

```
I'm going to ask about you height, feet and inches.
How many feet tall are you?
6
And how many inches?
2
You are 187.96 centimeters tall.
Press any key to continue . . .
```

Two more things to know for now (we'll discuss them in more detail later.) Any variable in C# can be converted to a string, you just add the .ToString() to the end of the variable. Second to combine two or more strings into one you can concatenate them, which means to put them together, by using the plus sign. That is how the line:

```
Console.WriteLine("You are " + cm.ToString() + " centimeters tall.");
```

Becomes

You are 187.96 centimeters tall.

By using the .ToString() on the cm variable and concatenating all of the strings together for the output.

(Note depending on the context the .ToString() isn't always necessary, but we'll always use it here for clarity.)

Finding the Gold Game

Next let's take the first steps to make our Console game "Find the Gold." The concept of this game is pretty simple; we'll have the player make a series of choices to guess which way to find a bag of gold. The first thing we'll do is ask for the player's name and print it out. Then we'll ask the player to choose a door, 1, 2, or 3 and for now just print out the door number. Go ahead and create a new Console project called FindGoldGame. Here is the code:

```
static void Main(string[] args)
{
    Console.WriteLine("Welcome to the Find the Gold Game!\n What is your
        name?");

    string name = Console.ReadLine();

    Console.WriteLine("Hello, " + name + ". Which door do you choose,  1, 2,
        or 3?");
```

```
    int door = Convert.ToInt32(Console.ReadLine());

    Console.WriteLine("You chose door " + door.ToString());
}
```

Here's an example output:
Welcome to the Find the Gold Game!
What is your name?
Curtis
Hello, Curtis. Which door do you choose, 1, 2, or 3?
2
You chose door 2
Press any key to continue . . .

This code should be straightforward. We just ask for the player's name and then ask for a door number. The only new thing is the '\n' character in the first WriteLine. The '\n' means newline, and it's our way of telling the text to hit return on the keyboard. That's why when we run the program the "What is your name?" appears on the second line. Then we print out the door number the player chose. In the next part we'll put in switches so the choosing the door will mean something.

Summary

This chapter took a look at simple variables in C#. These are the building blocks of all our programs, and in the next chapter we'll add to our programming capability by looking at how to change the direction the code executes in.

Chapter 3: Moving Around: Branching and Looping

Let's imagine a long listing of computer code, just a bunch of statements one after the other:

```
statement 1
statement 2
statement 3 ....
```

and so on. And let's imagine a computer executing the code and stepping through it directly, just going statement 1, statement 2, statement 3 .... The program can do a lot of things going through the code directly, but it could do more interesting things if instead of going one by one it could choose to go to different parts of the code. Maybe it goes statement 3, statement 4, then jumps to statement 7, statement 8 and continues. Or maybe it skips statements 7 and 8 and goes straight to 9. This concept of letting code run in different ways it called branching. Another thing we could have the program do is repeat a few lines of code, instead of going to statements 7, 8, 9, and then to 10 it could repeat 7, 8, and 9. It could execute 7, 8, 9, then repeat it again and again a few times, then go on to line 10. This concept of repeating a series of statements is called looping and is very important to programming. We'll cover branching in the first part of this chapter and looping in the latter part. Specifically we'll do the following.

- Learn about branching in code
- Implement the if-else statement
- Implement the switch statement
- Learn about looping
- Implement the for, while, and do-while loops

Branching

Branching in computer programming means being able to change the direction of code when it executes, skipping some parts of code and executing others based on certain conditions. This is a fundamental

concept of all structured programming languages and the most common way to branch code (in all structured languages not just C#), is to use an if statement.

if statement

The idea behind the if statement is simple: when the program goes to the if statement it performs a test. If the test is true the code in the block after the if statement is executed, if the test is false the code is skipped. The syntax is:

```
if( test )
{
    // Code here runs if test is true
}
```

This works great for many situations. But besides the single if block we can add an else block after the if. The else says that if the test is false we execute the code following the else. The syntax for this if-else statement is:

```
if( test )
{
    // Code here runs if test is true
}
else
{
    // If the test was false, code here runs instead
}
```

Let's make a more complete program that's pretty simple but will give an example of the if statement in action. Create a new Console project and add the following lines of code so the main function looks like the one below:

```
static void Main(string[] args)
{
    int age;
```

```
    Console.Write("Enter your age: ");
    age = Convert.ToInt32( Console.ReadLine() );

    if (age > 30)
    {
        Console.WriteLine("You're over 30.");
    }
    else
    {
        Console.WriteLine("You're not over 30 yet");
    }
}
```

Then run the program with Ctrl+F5. The program is pretty straightforward. The first few lines declare the variable age (which stores the user's age) that it gets from the user. Then the test "age > 30" is done in the if statement and tests if age is greater than thirty. If it is the user is told they're over thirty, if not they are told they're not.

The first question we can ask concerning the if statement is what kind of tests are there? There are several different types of tests we can do, but it all comes down to something that will output the values true or false. These types of tests are inspired by Boolean logic, a type of mathematical logic.

Boolean Tests

Remember in chapter two we discussed the boolean variable type, which can be set to true or false. Well this type can be used for our if tests. Here's a slightly contrived example of a program where at some point we need to decide to draw a triangle or not:

```
bool drawTriangle;

... // Some code happens to set drawTriangle to true or false

if( drawTriangle )
{
    // Draw the triangle
```

```
}
```

We can also use the not operator (an exclamation point in front of the boolean variable) to reverse the true/false for a test. For example:

```
if( !drawTriangle )
{
    // Code here runs if drawTriangle is false
}

// !drawTriangle means not true
```

Comparison Tests

Besides just tests with boolean variables directly we can do many comparison tests. The most basic tests are with two numbers and we can compare if one is less or greater than the other, just like in basic math. To write things mathematically we can say that for any two numbers A and B we can do the following tests:

```
if( A < B )     // True if A is less than B
if( A <= B )    // True if A is less than or equal to B
if( A > B )     // True if A is greater than B
if( A >= B )    // True if A is greater than or equal to B
```

This is the type of test we used in the age program previously. Besides these greater/less than ones we can also test for equality. To test if two numbers are equal we use the equality sign, which is two equal signs:

```
int A, int B;
// Some code does things with A and B
if( A == B )
{
    // Code here runs if A is equal to B
}
```

So in C# for "equals" we use two equal signs. Note too we use this equality test with integer type numbers, which are whole numbers, but usually not with decimal numbers. The problem with comparing decimal

numbers such as float for equality is that they can be unreliable because of precision problems and round off errors. Let's say we have:

```
float number;
...
// Here number = 0.000000001f
if( number == 0.0f )
{
    // This code is skipped, since number doesn't equal zero
}
```

But in the above case we probably wanted the code inside the if brackets to be executed, since number is very close to zero, but we could easily have a round off error that caused an extra 1 at the end of it that throws the test off. And the equality tests aren't just for numbers, many classes have built in the == for an equals test. For example we could compare if two string are equal with it.

And with the equality test we can also have a test for not being equal. The symbol for this is an exclamation point and an equals sign !=:

```
int A, int B;
// Some code does things with A and B
if( A != B )
{
    // Code here runs if A is not equal to B
}
```

Remember the exclamation point ! means not true.

The previous tests for the if-statements all used a single test to decide if the code would run or not. But inside the if statement we can combine multiple tests. We do this by using the and, &&, operator and the or, ||, operator.

The and operator says that for the test to be true, all of the conditions must be true. Let's say that we wanted to test if someone's age is between

20 and thirty. This is two tests, one if the age is 20 or more and another if it is less than or equal to 30. We make sure both of these are true by "anding" them together:

```
if( age >= 20 && age <= 30 )
{
    // Code runs here if age is between 20 and 30
}
```

Now let's say that instead of identifying people between twenty and thirty, we want to identify people who are either pretty young, less than 15, or pretty old, over 85. So we want our if statement to be true if the age is under 15 or over 85, then we'll put an OR, | |, between them (Note: The vertical slash symbol can be hard to find on the keyboard for a lot of people at first, it's the capped of the backslash \ in the upper right corner of the keyboard.) Our if statement will then look like:

```
if( age < 15 || age > 85 )
{
    // Code here runs if age is under 15 or over 85
}
```

The OR statement will also be true if either of the conditions is true. Unlike the "or" used when were speaking, which usually means just one or the other. For instance, if someone said "I'm going to the grocery store today or tomorrow" we take this to mean that they will go to the store today or tomorrow, but not go the grocery store both today and tomorrow, which is what the computer OR means. As an if statement it would be:

```
bool gotoStoreToday, gotoStoreTomorrow;

if( gotoStoreToday || gotoStoreTomorrow )
{
    // Code here runs if gotoStore today is true, or if
    // gotoStoreTomorrow is true,
    // or if both of them are true
}
```

That kind of one or the other OR is called exclusive or, XOR, but it isn't used with these types of test in C#.

Combining Tests

We can combine multiple ANDs and ORs to make tests as complicated as we want. To keep them clear like in mathematics we put parentheses around tests to say which ones go first. Let's say that for whatever reason we are making a program that runs some special code for weekends in December, so we need to test in its code if today is Saturday or Sunday and if the variable month is December.

```
if( (today == Day.Saturday || today == Day.Sunday) && month ==
Month.December )
{
    // Code here runs if it's a weekend in December
}
```

The Day.Saturday and Sunday and Month.December are enums, a special way to write values for variables to make them more readable. We won't worry about that for now, let's just look at the tests inside parentheses in the if statement, it tests true if today equals Saturday or Sunday. Then the overall test is true if month is also equal to December. We could add more and more conditionals and these types of tests can become long and complex, but usually they'll be simple, just one or two conditionals.

More on if

Now that we have a pretty good understanding of the kinds of tests we can do inside of the if statement and the basic syntax of it, let's look at a few more little points about the if statement. For starters, if we just want one line to execute if the if is true (or in the else block) we don't need parenthesis, without parenthesis only the first line after the if is run.

```
if( testIsTrue )
    success = true; // Just one line after if
else
```

```
    success = false;
```

Nested ifs

Another issue with ifs is that we can nest them, put then inside of each other.

```
if( today == Day.Monday )
{
   if( time < 12.00 )
   {
    // Code here runs if it before 12 on Monday
   }
   else
   {
    // Code here runs if is after 12 on Monday
   }
}
```

These nested if's can be helpful for adding more branches and ways to go inside of our code. The only tricky thing to watch out for with the nesting is the else. The else must be set based on where it is in the parenthesis or it matches up with the previous if. For example:

```
if( test1 )
    // Test 1 is true doe line
if( test 2 )
    // Test 2 is true do line
else
    // This line runs if test 2 is false,  it doesn't have anything to do with test 1
```

Besides nesting another common thing to do with if statements is to put them in a series. Here's an example:

```
if( test1 )
    ...
else
if (test 2 )
{
    // Code here runs if test 1 is false and test 2 is true.
}
```

We could keep adding these else-if's to the end of the if-statement. This is a good way to choose among several options. If we have too many of these, though; we might want to use a switch statement instead.

Switch Statement

A switch statement takes in a variable and then does different code based on the value of the variable. This is helpful when we have a series of tests based on a single variable. Here's an example:

```
static void Main(string[] args)
{
   int number;

   Console.Write("Enter a number between 1 and 5 : ");
   number = Convert.ToInt32(Console.ReadLine());

   switch (number)
   {
        case 1:
            Console.WriteLine("One is a good number");
           break;
        case 2:
           Console.WriteLine( "Two is second best" );
           break;
        case 3:
           Console.WriteLine("Three is something");
           break;
        case 4:
        case 5:
           Console.WriteLine("Four and Five aren't bad");
           break;
         default:
        Console.WriteLine("You didn't enter a number from 1-5");
           break;
   }
}
```

The switch statement works by stating a variable in parenthesis after the switch:

```
switch( number )
```

Then following this it decides which code to execute based on its value. In the brackets after the switch we have a number of cases. They take the format:

```
case value:
```

So for our int variable we use the case:

```
case 1:
// Code to execute if variable is the value 1

break; // Stop executing code
```

Each case in our number example is just a different value for the number, if the value is true the code after the : is executed until a break statement is hit. If we want to have two or more cases execute the same code we can just put two cases on top of each other, just like the case 4: and case 5: in our example. The special case of default: at the end is just what it says. If we went through the switch statement and never had a correct case we'll run the code after the default. The cases are just like multiple if's.

The switch statement isn't used nearly as much as the if statement. It's usually reserved for special cases where if we have a whole bunch of different choices of code based on the value of just one variable. A use that will come up for it a lot is when we read a variable for a key being pressed at the keyboard, and we might do different code depending on all the different keys that could have been pressed.

This covers the two types of branching. Next we'll look at looping.

Looping

For Loop

Many times in programming we need to repeat a block of code several times. One of the ways to do this is a for loop. A for loop uses a counter variable. It starts by initializing the counter variable, performs an operation on the counter at each loop, and runs until the counter variable passes some condition (test). This might be easiest to see with an example, we'll add the following after the else statement above:

```
Console.Write( “Enter your age: “ );
int age = Convert.ToInt32(Console.ReadLine());
Console.Write("Enter the current year: ");
int year = Convert.ToInt32(Console.ReadLine());

for (int i = 0; i < 5; i++)
{
   year++;
   age++;
   Console.WriteLine( "In the year " + year.ToString() + " you'll be " +
age.ToString() + " years old." );
}
```

An example run of this program is:

```
Enter you age: 105
Enter the current year: 2009
In the year  2010 you'll be 106 years old.
In the year  2011 you'll be 107 years old.
In the year  2012 you'll be 108 years old.
In the year  2013 you'll be 109 years old.
In the year  2014 you'll be 110 years old.
```

This addition prints out the user's age for the next five years. The for loop starts by creating the counter variable i and setting it to zero, and will continue to loop while i is less than five, and at each loop run it adds one to i.

Notice that it is common in programming to count using zero based indexing which means instead of starting counting at 1, such as 1,2,3.. we start at zero 0,1,2 ... If we had three objects, the first would be numbered 0, the second would be numbered 1, and the third would be number 3. So to count to five we start with i = 0 and it counts 0,1,2,3,4 – which is five loops total. This starting at zero can take a little getting used too, but it's good to get into the habit of counting this way since when we get to arrays we'll see you'll have to use zero-based indexing.

We see that the for loop uses the counter variable and consists of three parts.

In the first part we initialize the counter variable, usually just setting it to zero, as in our int i = 0;

In the second part we define an exit condition for the for loop. Exit conditions are important in looping, an ext condition is just a way to make sure the loop stops at some point and doesn't repeat forever. The exit condition is a statement that says the loop is over. Usually with for loops our exit condition is just testing if our counter variable has gone up to a certain number.

The third part we perform an operation on the counter variable. This is usually just adding one to the counter. The syntax for these three things is:

```
for( counter variable initialization; counter variable test; counter variable operation )
```

Note the placement of the semicolons, its important to have them in the proper places for them to work correctly.

Multiple for loops

We can have multiple for loops together, this can be helpful as we go through multiple things at once. Here's a little example that will print out a little 10 by 10 grid of numbers.

```
static void Main(string[] args)
{
    for (int i = 0; i < 10; i++)
    {
        for (int j = 0; j < 10; j++)
            Console.Write(j.ToString());
        Console.Write("\n");
    }
}
```

The output or this program is:

```
0000000000
1111111111
2222222222
3333333333
4444444444
5555555555
6666666666
7777777777
8888888888
9999999999
```

For loops are very common and occur in every procedural language, such as C++ and Java, they are handy for running code a specific number of loops. But what if we don't know how many times we want the loop to run? The loop we can use instead of the for loop is the while loop.

while loop

A while loop is another loop type that is simpler than the for loop. All that is needed for a while loop is have a conditional, test, and while that test is true the while loop will keep executing. The basic format is simple:

```
while( test )
{
    // Test is true, execute code here
} // End of while loop, go back up to test
```

Let's create a small example, a program that asks the user to guess a number, and then keeps asking (looping) for another guess until the user guess the correct number:

```
static void Main(string[] args)
{
    int secretNumber = 3;

    Console.WriteLine("Guess the number game, enter a number between 1
        and 10: ");

    int guess = Convert.ToInt32(Console.ReadLine());

    while (guess != secretNumber)
    {
        Console.WriteLine("Wrong, guess again : ");
        guess = Convert.ToInt32(Console.ReadLine());
    }
    Console.WriteLine("Correct!");

}
```

A sample run of this is:

```
Guess the number game, enter a number between 1 and 10:
5
Wrong, guess again:
3
Correct!
```

This program picks a secret number and will continue looping while the number inputted does not match it. (Remember != means not equal.) This program could loop 10 times 100 times, or a 1000 times (as long as three is not guessed.) The one thing to watch out for is that the loops have an exit condition, a guarantee that the loop will not run forever. If we wrote a loop that had no way of exiting, such as:

```
static void Main(string[] args)
{
    while (true)
    {
        Console.WriteLine("We're looping forever ");
    }
}
```

It will never end. When this happens in the best cases this just hangs up the program. In others it causes everything to crash. We always want some way to exit. Besides just the initial test condition, there are two other ways that we can control how the loop runs, the break and continue statement. Both only work inside of loops, and work for for's and while's. The continue statement says to ignore everything below it in the run of the loop, and go back to the beginning. For instance, look at the following code:

```
int test = 0;
while( test < 10 )
{
if( test == 5 )
        continue;

    Console.WriteLine( test.ToString() );
     test++;
}
```

The output for this is:

012346789

The above program (which is similar in it's working as a for loop) prints out the numbers 0 to 9, with the exception of 5. The if test is equal to 5 the break happens and the rest of the loop is skipped and the loop goes back to the beginning.

The break statement simply exits the loop directly. The following loop will print out the numbers 0-9, though the test on the while loop is always true the loop will exit when the test number goes above 9:

```
int test = 0;
while( true)
{
    if( test > 9 )
        break;
    Console.WriteLine( test.ToString() );
    test++;
}
```

The output is simply:

0123456789

When test is over 9 the break statement is called and the loop exits.

do-while

Let's go back a little bit to our secret number guessing game.

```
int guess = Convert.ToInt32(Console.ReadLine());

while (guess != secretNumber)
{
    Console.WriteLine("Wrong, guess again : ");
    guess = Convert.ToInt32(Console.ReadLine());
}
Console.WriteLine("Correct!");
```

In this program if the user enters the correct number the test for the while loop (guess != secretNumber) evaluates to false on the first try, and all of the code inside the while loop is skipped. Let's say we wanted the code inside a while loop to execute at least once, no matter what. For this we use a do-while loop, it always executes at least once because the test comes at the end of the loop. For example let's look at a different (and worse) way we could have written the guessing number program.

```
static void Main(string[] args)
{
    int secretNumber = 3;

    Console.WriteLine("Guess the number game");
    int guess;

    do
    {
        Console.WriteLine("Enter a number between 1 and 10:  ");
        guess = Convert.ToInt32(Console.ReadLine());
    } while( guess != secretNumber );

    Console.WriteLine("Correct!");

}
```

A sample run of this is:

```
Guess the number game
Enter a number between 1 and 10:  5
Enter a number between 1 and 10:  3
Correct!
```

This program also keeps looping until the correct number is guessed, and the loop will be entered at least once.

Goto statement

Just a little note, there is one more kind of branching that we can do, and that is the use of the goto statement. The goto is an older method to jump around code, you mark different points of the program and then use goto to jump around to those points. We won't cover it here as we really shouldn't be using this, it quickly creates "spaghetti code," code that jumps around a lot and is mostly unreadable.

Finding the Gold

When we left our "Finding the Gold" game we had put in the functionality to let the player enter their name and choose a door:

```
Console.WriteLine("Welcome to the Find the Gold Game!\n What is your
    name?");

string name = Console.ReadLine();

Console.WriteLine("Hello, " + name + ". Which door do you choose, 1, 2, or
    3?");

int door = Convert.ToInt32(Console.ReadLine());

Console.WriteLine("You chose door " + door.ToString());
```

Now let's add in three choices. If the player chooses door 1 we'll let the player then choose to open box one or two. If the player chooses one we'll tell them it's empty, if they choose door two we'll tell them they found a bag of gold. So after the above add the following:

```
if( door == 1 )
{
    Console.WriteLine("You found two boxes, which do you choose?");

    int box = Convert.ToInt32(Console.ReadLine());

    if( box == 1 )
        Console.WriteLine("It's empty. You lose!");
    if( box == 2 )
        Console.WriteLine("You found a bag of gold!");
}
```

The output of this part of the program is:

```
You choose door 1
You found two boxes, which do you choose?
2
You found a bag of gold!
```

Next we'll have it if the player chooses door two we'll assume they fell into a hole. We'll print "you're falling" five times:

```
if( door == 2 )
{
    for( int i = 0; i < 5; i++ )
        Console.WriteLine("You're falling!");
}
```

The output of this part of the program is:

```
You choose door 2
You're falling!
You're falling!
You're falling!
You're falling!
You're falling!
```

As an option for door three you could tell the player the gold is in a safe have them need to undo a combination to get to it. Then you could insert the guessing game code to have them guess a number to find the gold.

Summary

Now we are getting closer to be able to do really interesting things in code. We looked at branching using the if-else and case statements, and how to create loops with the for loop and the different whiles. In the next chapter we'll look at some more advanced data types and how to divide up code with functions.

Chapter 4: Advanced Data Types and Functions

What we've gone over in the preceding chapters may not have been super exciting, but we're well on our way to having the skills to be a game programmer. We've looked at how programs are organized, simple variables, operators and a few other instructions, and branching and looping. In this chapter we're going to go to a more advanced level. We'll make our code more powerful by going over some more complex, advanced data types: strings, arrays, lists, and structs. Then we'll look at ways to modularize code by creating functions to put blocks of code in.

- Learn about strings and how they work in C#
- Learn how to use arrays and collections
- Learn how to use structs
- The basics of functions

Strings

We discussed strings in the second chapter, but since they are so important we'll talk a bit about them again here. One of the most basic things to deal with in any programming language are strings. Strings, if you recall, are just lists of characters. This sentence is a string, even this entire book is a string, just a whole bunch of characters. Since we are all about manipulating data in computer programs and data is usually in the form of strings, they are very important. Pictures, such as bitmaps, are stored as strings, just long sequence of characters containing numbers for each color of every pixel. 3D Models likewise are just long strings that contain various kinds of information about the model. Strings are a major data structure in programming.

Strings have always been important in programming, but before .Net working with strings could get a bit too complex. Back in the days of C all of the basic string functionality was about just simple blocks of memory and some basic functions for manipulating them. C++ brought

better strings that could do more things and a distinct string class was created. But these strings still weren't enough for a lot of programmers, so different people started making different string classes and functions of their own. Converting and keeping track of all these different strings wasn't simple and unnecessarily confusing. Instead of all that trouble in the .Net framework we're given a single robust and good String class to work (which is C#'s string variable type.) The .Net string class has a lot of functionality, and there are many helper functions to turn strings to something else (such as numbers) and back. This single string class is one of the major components of C# and .Net.

To declare a string we just use the string keyword:

```
string stringName;
```

Where stringName is the name we want to call that string. Remember actual strings are put in double quotes:

```
string helloString = “Hello”;
```

Like we said, strings in C# are just another way of using the String class in .Net. The nice thing about this is that if we ever need to program in a different .Net language, such as Visual Basic.NET or Managed C++, the strings will work just the same. If you do a search for Strings in the Visual C# Express documentation you'll find that there are many things you can do with strings. We'll look at just a few of the most commonly used ones.

Concatenation

One of the most basic operations we can perform on strings is to combine two strings into one. The most common way to do this is if we have two strings to just put the second string at the end of the first one. This attaching of strings one at the end of the other is called concatenation. Here is an example:

```
static void Main(string[] args)
{
    string name;
    Console.WriteLine("Hello, what's your name?");
    name = Console.ReadLine();
    string outString = "Hello, " + name + ", nice to meet  you.";
    Console.WriteLine(outString);
}
```

A run of the program looks like this:

Hello, what's your name?
Curtis
Hello, Curtis, nice to meet you.

The first part of this program declares a string called name and then reads the user's name into it. The third line of the program:

```
string outString = "Hello, " + name + ", nice to meet you.";
```

concatenates the strings "Hello", name, and "nice to meet you" and stores the new string in the variable outString. The plus symbol (+) is used as the concatenation symbol. And like the unary addition shortcut we can do the following to concatenate:

```
string helloString = "Hello";
helloString += ", nice to meet you";
// helloString now has "Hello, nice to meet you"
```

Concatenation is our way of combining strings.

(Note doing concatenations on the string class is not very efficient and the .Net framework has a special StringBuilder class that does them better, but we won't worry about that here.)

Comparing

Another simple thing to do with strings is to compare if two are equal. Strings can be compared in conditional tests using the == sign and != (not equals) sign similar to numbers. Here's an example:

```
static void Main(string[] args)
{
    string name= "Charlie";
    if (name == "Charlie")
        Console.WriteLine("The name here is Charlie");
    if (name != "Mike")
        Console.WriteLine("The name is not Mike");
}
```

The output for the above is:

The name here is Charlie
The name is not Mike

This test is case sensitive, so the following test will be false:

```
string name = “Charlie”;
if( name == “charlie” )
    // This test is false
```

To ignore case sensitivity we can use the .ToUpper() or .ToLower() methods on the strings. These methods convert a string to all upper case or lower case. The following test will be true:

```
string name == “Charlie”;
if( name.ToLower() == “charlie”.ToLower() )
    // This test is true
```

Converting

Like we mentioned earlier in the book, it's easy to convert strings to and from other data types. Let's look first at converting numbers to strings.

```
int goodNumber = 5;

string outString = “I thing a good number is “ + goodNumber.ToString();
```

This program converts the goodNumber integer to a string. This isn't anything special for integers, all variables and data structures in C# can be converted to a string this way. Anything at all in C# (and the .Net framework) that you want converted into a string just put the .ToString() after it. Going the other way (which we saw at the end of chapter two)

```
int goodNumber;

Console.WriteLine("Enter a good number");
string numberString;
numberString = Console.ReadLine();
goodNumber = Convert.ToInt32(numberString);
Console.WriteLine(goodNumber.ToString() + " is a good number.");
```

Here's an output:

```
Enter a good number
25
25 is a good number
```

If we run this program it will just ask the user for a number and then print it back out. The first few lines here just have the user enter an integer and store it in the string numberString. The line:

```
goodNumber = Convert.ToInt32(numberString);
```

changes our string into a an integer. As you type this line in notice that after typing the period at the end of Convert a whole list of options comes up of things we could try to change the string into, such as ToBoolean() (to convert to a bool) or ToSingle() (which is used to convert to a float). The reason the names of the types in the Convert don't match up exactly in name to our C# data types is that the Convert is part of the .Net framework, not C# specifically, so the names for the data types are the .Net names, not the C# names.

Now trying to run the program again instead of entering a number enter the string "hello". You'll get the following:

```
Enter a good number
hello

Unhandled Exception: System.FormatException: Input string was not in a
correct format.
   at System.Number.StringToNumber(String str, NumberStyles options,
NumberBuffer& number, NumberFormatInfo info, Boolean parseDecimal)at
System.Number.ParseInt32(String s, NumberStyles style, NumberFormatInfo
info)at StringTest.Program.Main(String[] args)
```

Entering a string that won't convert to an integer creates an exception, an error that popups up when running the program that basically says something went wrong. Exceptions are called runtime errors, since they occur when the program is running. These are different than compile time errors, errors that occur from compiling the program. We won't worry about handling exceptions here, for now just be careful about converting strings.

Concatenation, comparing, and converting are three important things we can do with strings. Again there are a lot more things you can do with them (you can check the documentation) but we'll move on to our next advanced data type, arrays.

Arrays

Let's say that we're making a space ship game (which we'll do in the next part) and in it we have several ships and we need to keep track of the speed of each one. We could try creating float variables for each ship's speed:

```
float ship1Speed;
float ship2Speed;
...
float ship58Speed;
```

But this would quickly get annoying, and would take up a lot of code space having them all typed in. It would be nice if we could create a single variable that holds multiple values. That is what an array is, an array is a variable that holds a list of values of the same type. We mark that a variable is an array by using straight brackets, [] and the individual values stored in an array are called the elements or the members of the array. To create an array to hold all of out ship's speeds we would just say:

```
float[] shipSpeed;
```

This says to make a variable shipSpeed that holds multiple values of floats in a list. Arrays are a little bit trickier than usually variables. So let's look at a few important aspects of them.

Initializing arrays

There are a few ways we can set what's in an array initially and its size (how many members it has.) The first way is to declare a list of elements to put in it when we declare the array. To create a shipSpeed array with that holds five values, 100 to 500, we would just say:

```
float[] shipSpeed = { 100.0f, 200.0f, 300.0f, 400.0f, 500.0f }
```

But what if we didn't know what values we wanted to store in the array? Instead of listing them out we can just give a number of variables to store, a size of the array:

```
float[] shipSpeed = new float[5];
```

The above line creates an array with spots to hold five floats. The new is used because an array is what is called a reference type. The precise meaning of this is a little technical. It basically says that the variable itself (shipSpeed) doesn't store all the floats in memory itself, it just contains an address to a memory location, a reference to some spot in memory. So

when we use a reference type we put the new keyword in front of it to create the location in memory for it. (If that explanation doesn't make sense don't worry about it, for now just think that for reference types (like an array) we sometimes have to use new when defining the variable.)

Accessing members

We just saw how to create the array, but how do we access and modify the individual members of the array? We simply in the brackets following the array put the number of the element we want.

```
float[] shipSpeed = { 100.0f, 200.0f, 300.0f, 400.0f, 500.0f }
float speed1 = shipSpeed[0]; // We just put 100.0f in speed1
shipSpeed[1] = 25.0f; // Now the value of the second element is 25.0f instead
// of 200.0f;
```

One thing to notice here is that to access the first element instead of using 1 we started with 0. This is because arrays use zero based indexing. As we mentioned before this means that when we count things we call the first thing 0, the second thing 1, and so on. So if we had, say, four elements, we would count them 0,1,2,3. This can be a bit unusual at first when working with arrays to think like if we want to change the third element we refer to it as number 2, but after a while it becomes natural.

What if we try to access a member of an array and specify a number too but or small for it, like the following:

```
float[] shipSpeed = { 100.0f, 200.0f, 300.0f, 400.0f, 500.0f };
float speed1 = shipSpeed[12];
```

In this program we try to access the thirteenth element (number 12) which doesn't exist. If you run it you'll get another exception. So you have to be careful when accessing array to make sure you are in range. To help with this you can call the .Length function on an array to see how many are in it:

```
float[] shipSpeed = { 100.0f, 200.0f, 300.0f, 400.0f, 500.0f };
int size = shipSpeed.Length; // size is the number 5
```

Iterating an Array

Sometimes when working with arrays we want to do something to every element in the array. If we go through every element in an array and do something to it (such as assign them a value) we are iterating through the array. To do this we can use a for loop. Let's assign the numbers 0, 100, 200 … to ten ship speed elements:

```
for( int i = 0; i < shipSpeed.Length; i++ )
    shipSpeed[i] =  i * 100;
```

Besides using the regular for loop C# also has a special kind of for loop, a foreach loop. The foreach is just a simple way to go through each element of an array as long as we don't modify it. Say we want to print each ship's speed. We would just code the following:

```
foreach( float speed in shipSpeed)
    Console.WriteLine(speed.ToString());
```

The syntax here is pretty simple to figure out, we just specify the type and the array to iterate through, then we can use every element.

Multi-dimensional arrays

Besides having arrays that hold a single row, list, of data we can make rectangular arrays; arrays that hold multiple rows of data. The following will create three rows with each row having two floats in it:

```
float[,] rectangleArray = new float[3,2];
// Assign an element:
rectangleArray[1, 0] = 5.0f;
```

This creates a rectangular array, but we can also make jagged arrays. That is an array where each element is itself an array. This way the multidimensional array doesn't need to be rectangular, each array can be of a different size (that's why it's called jagged.) Here's an example:

```
float[][] jaggedArray = new float[3][];

// Give row zero an array of 2 elements
jaggedArray[0] = new float[2];
// Give row one an array of 3 elements
jaggedArray[1] = new float[3];

// Assignment:
jaggedArray[0][1] = 5.0f;
```

Collections

Arrays are the most common way in computer languages to handle lists of elements. But in C# we other methods of doing this called Collections. One of the most common collections is called a list. It is like an array except we don't have to worry about setting a specific size to it, we just add and remove elements to it as we see fit. Here is an example that makes a list of numbers:

```
List<int> integerList = new List<int>(); // Make a list of int type
integerList.Add(5); // Add the number 5 to the list
integerList.Add(6); // Add the number 6 to the list
int firstInt = integerList[0]; // firstInt now has the value 5
```

We'll being lists more in the 2D part. The syntax for them is like the arrays. The big difference is that for lists we add items as we go. We can see where this will be useful when we keep a list of items that we add too periodically through the game. In the next part in the 2D game as we fly our ship and we need to create new enemies we will create them and add them to an enemy ship list.

Structs

Moving on from arrays let's say that in our ship game we wanted to store more information about a ship than just its speed. We also want to store a ship id which is an integer, and a ship type which is a string. We could create three separate variables for each ship and just note that they are all describing the same object, but it would be easier if we had a structure that did this for us. C# does give us just that, a struct, which holds multiple variables together. A struct for the ship variables just described would be:

```
public struct ship
{
    public int id;
    public float speed;
    public string type;
}
```

The basic syntax is just

```
public struct structName
{
    public variableType1 variableName1;
    public variableType2 variableName2;
    ...
}
```

We put the name of the struct first and then in the brackets after it we put a list of the variables in it (we put public in front of each variable to make it accessible, we'll go into more detail about this in the next chapter.) To use the struct we declare a variable of our struct type, and to access individual members (variables in it) we use a period to specify it. All of this is easiest seen with an example:

```
public struct ship
{
    public int id;
    public float speed;
    public string type;
```

```
}

static void Main(string[] args)
{
    ship myShip; // Just created a ship structure
    myShip.id = 33; // Give it an id of 33
    myShip.speed = 2200.0f; // Give it a speed of 2200
    myShip.type = "class 1 ship"; // Give it a type

    Console.WriteLine( "My ship's type is " +    myShip.type.ToString() );
    Console.WriteLine( "Its speed is " +  myShip.speed.ToString() + " and its
    id is " + myShip.id.ToString() );
}
```

The output is:

```
My ship's type is class 1 ship
Its speed is 2200 and its id is 33
```

This program uses the ship struct to create a variable of the ship type (myShip) and then assigns each of its variables a value. You might notice that this grouping of variables together to describe a specific object (in this case the ship) sounds like Object-Oriented programming. Structs are the beginning of Object-Oriented programming; while they are in structured languages like C, are like objects. The difference is that structs usually contain just variables while classes (objects) contain variables and functions (actions to act on the data)

Example Program

Strings, arrays, and structs are three advanced data types, and while we went over each of them, let's make a simple example program that uses all three together. The program, called shipList, will store an array of ship structs and the user will enter info in on each ship and then it will print out a summary of everything that was entered. Here's the code:

```
public struct ship
{
    public int id;
```

```
    public float speed;
    public string type;
}

static void Main(string[] args)
{
    // Write out what this program is
    Console.WriteLine("Ship inventory program.");

    // Get the total number of ships to record
    Console.Write("Enter how many ships do you want to record: ");
    int totalShips = Convert.ToInt32(Console.ReadLine());

    // Create a ship array
    ship[] shipArray = new ship[totalShips];

    // Go through each ship and get some info on it
    for (int i = 0; i < shipArray.Length; i++)
    {
        Console.Write("Enter id of ship : ");
        shipArray[i].id = Convert.ToInt32(Console.ReadLine());

        Console.Write("Enter speed of ship : ");
        shipArray[i].speed = Convert.ToSingle(Console.ReadLine());

        Console.Write("Enter type of ship : ");
        shipArray[i].type = Console.ReadLine();
    }

    // Now print out the info on each ship
    Console.WriteLine("Ship info: \n");
    foreach (ship shipElement in shipArray)
    {
        string printString = "Ship id: " + shipElement.id.ToString() + "\n";
        printString += "    type is " + shipElement.type + "\n";
        printString += "    speed is " + shipElement.speed + "\n\n";
        Console.WriteLine(printString);
    }
}
```

An example run of this program would be:

```
Ship inventory program.
Enter how many ships do you want to record: 2
```

```
Enter id of ship : 33
Enter speed of ship : 2000
Enter type of ship : Class I
Enter id of ship : 45
Enter speed of ship : 2200
Enter type of ship : Class II
Ship info:

Ship id: 33
   type is Class I
   speed is 2000


Ship id: 45
   type is Class II
   speed is 2200
```

The comments describe what's going on in each part of the code. A couple of points: The array itself is an array of the ship type. Notice that the user chooses the size of the array (when entering the totalShips data.) Like we said the size of the array can be determined at run-time (when the program is running) instead of compile-time (when the program is being compiled.) We also use the escape key "\n" to create new lines when we need them. Whenever we use Console.WriteLine we can create new lines by inserting \n.

Functions

What we can do in our programs is getting more advanced, let's keep moving and go over how functions work in C#. Functions, if you recall from chapter 1, are blocks of code set aside that can be called to run when needed. The concept is pretty simple, when writing code one of our goals should be never to duplicate code, that is we don't want to write the same block of code twice in the same program. There are a lot of reasons for this, not duplicating code saves space, reduces errors (since it keeps the total amount of code down), and is easier to make changes (when we need to change code we only have to change it at one spot.) So we never want to duplicate code, and functions are a way to do that.

Let's look at a simple example of a function and then go through it line by line:

```
class Program
{
    static void printHello()
    {
        Console.WriteLine("Hello everybody!");
    }
    static void Main(string[] args)
    {
        printHello();
        Console.WriteLine("I said,");
        printHello();
    }
}
```

The output of this is

```
Hello everybody!
I said,
Hello everybody!
```

The first part of this program defines the function:

```
static void printHello()
{
    Console.WriteLine("Hello everybody!");
}
```

The name of the function is printHello, and we can tell it is a function because it has the parenthesis () following it. The static void in front of printHello, are descriptions of the function. We won't worry about what the static means for now, we'll deal with it in the next chapter. The void tells us a return-type, and we'll cover that briefly.

Inside of Main, which is a function itself, we tell the printHello function to run, we do this by using the name of the function as an instruction. Each

```
printHello();
```

Tells the compiler to go to the printHello function and do all of the computer code in it. In our program every time we say printHello() the text "Hello everybody!" is printed on the console, because the code direction goes to the printHello function.

Passing Variables

Putting code into little blocks of functions is good, but what about variables. If we put a variable in our Main function and then call another function the other function won't have access to the variable. For example, if you try running the following code you'll get an error message:

```
class Program
{
    static void printNumber()
    {
        Console.WriteLine("Printing number: " +
        number.ToString()); // Error – number isn't here
    }
    static void Main(string[] args)
    {
        int number = 5;
        printNumber();
    }
}
```

The error message says "number does not exist in the current context" referring to the printNumber function. Number is in Main, not printNumber, so printNumber can't use it. One solution to this is to make number a global variable, which means that every function in the whole program can access it. To make the number variable global we put it outside of our functions:

```
class Program
{
```

```
    static int number;

    static void printNumber()
    {
        Console.WriteLine("Printing number: " + number.ToString());
    }
    static void Main(string[] args)
    {
        number = 5;
        printNumber();
    }
}
```

(The static in front of the number can be ignored) This program will compile and run just fine, but we don't want to make variables global if we don't have to. Instead of doing this what we can do is pass the value of number to printNumber. We can pass the number to it like this:

```
class Program
{
    static void printNumber( int numberToPrint )
    {
        Console.WriteLine("Printing number: " + numberToPrint.ToString());
    }
    static void Main(string[] args)
    {
        int number = 5;
        printNumber(number);
    }
}
```

The output:

```
Printing number: 5
Press any key to continue . . .
```

In this program we took the number variable and passed it into the function. Our function knew what variables to take because we made a parameter list for it. The format for this is:

```
void functionName( type varName, typevarName, ... )
```

We can specify as many variables as we like to pass. Note that we're passing the value of the variable, not the variable itself. For instance the following program:

```
class Program
{
    static void printNumber( int numberToPrint )
    {
        numberToPrint = 7;
        Console.WriteLine("Printing number: " +
            numberToPrint.ToString());
    }
    static void Main(string[] args)
    {
        int number = 5;
         printNumber(number);
         Console.WriteLine("Printing number: " + number.ToString());
     }
}
```

Will have the output:

```
Printing number: 7
Printing number: 5
```

Changing the number to 7 in the function didn't affect the number in Main.

Returning a value

If we want to let a function change a variable we can pass it in by reference. To do this we add the ref keyword in front of the variable in the function parameter list.

```
class Program
{
    static void printNumber( ref int numberToPrint )
    {
        numberToPrint = 7;
```

```
        Console.WriteLine("Printing number: " +
            numberToPrint.ToString());
    }
    static void Main(string[] args)
    {
        int number = 5;
        printNumber(ref number);
        Console.WriteLine("Printing number: " +
        number.ToString());

    }
}
```

This program has the output:

Printing number: 7
Printing number: 7

Since we put a ref in front of the variable when passing it the actual variable place in memory was passed, and the number itself changed.

So if we want to have the function modify some data for use in other parts of the program we can pass it by ref. Besides passing variables by ref a better way to get data from a function is to have the function return a value. The same way we can pass data into a function we can also have a function pass it out. We do this by specifying a return type; the void in front of the functions we've written so far is actually a way saying we're not returning anything. The void means nothing. We can replace the void by a data type. For instance:

```
class Program
{
    static int printNumber( int numberToPrint )
    {
        numberToPrint = 7;
         Console.WriteLine("Printing number: " + numberToPrint.ToString());
         return numberToPrint;
    }
    static void Main(string[] args)
    {
```

```
        int number = 5;
        number = printNumber(number);
        Console.WriteLine("Printing number: " +  number.ToString());

    }
}
```

This program has the output:

Printing number: 7
Printing number: 7

Our function printNumber has a return type of int:

```
static int printNumber( int numberToPrint )
```

The function has in it the keyword return, which says to stop the function and return that value:

```
return numberToPrint;
```

Anything after the return is ignored by the compiler. If you change the printNumber function to:

```
static int printNumber( int numberToPrint )
{
    numberToPrint = 7;
    Console.WriteLine("Printing number: " + numberToPrint.ToString());
    return numberToPrint;
    Console.WriteLine("I'll never print");
}
```

That last WriteLine will never show up on the screen.

Summary

So we've got the basics of how functions work and before that we looked at a few more advanced data types: strings, arrays, and structs. Stucts are a precursor to objects in Object Oriented programming, so let's move on to the next chapter where we start making objects in our program.

Chapter 5: Object-Oriented Programming

In the first chapter we briefly mentioned the Object-Oriented programming (OO) paradigm. We talked about how C# is an Object-Oriented language. C# is actually what is called a pure Object-Oriented language, in that everything is an object. In C# the number 5 is an object, the different art resources, and everything we work with is an object. In this chapter we'll take a closer look at Object-Oriented programming. We will cover:

- The main concepts of OO programming
- How OO programs are designed
- How to create and use classes of objects

Object-Oriented Programming 101

Objects are everywhere in life. When using the term object here it means the same thing as in "real" life. This book is an object, the computer you work with C# on is an object, etc. Object-Oriented programming is an attempt to take these objects we find in the real world and translate them into computer code. This works for most types of programming, but really makes sense for game programming. When making games it's easy to think of things in terms of objects, since our games are full of "real" objects. Like in the example we mentioned in chapter one with the racing game; in making a racing game we can have the car be an object, opponents be objects, the terrain be an object, etc.

This thinking of the game in terms of objects is helpful, but how do we write the code to create objects? What we'll be doing to create objects is to create classes. Classes are templates for objects, they define the different attributes, variables and functions the object has. Classes are not objects themselves; they are descriptions for types of objects. Similar to the animal kingdom phylum, where classes are used to describe type of

animals, like there is a class of animals called lions. But an individual lion, such as the lion at the local zoo, is a lion object while the class lion is not an individual object. The actual objects, like the lion at the zoo, are called instances of an object. Classes are the heart of Object-Oriented programming; all of our coding in C# will be in defining classes.

Classes are a combination variables and functions (the functions are called methods.) The variables and methods of a class are called the members of the class. OO programming works by taking an object in the real or game world we want to model (instead of a car let's use the example of a spaceship) then defining it in terms of attributes, variables holding information about it (like it's speed, position), and what the object does (like slow down, turn left, turn right).

Classes are usually defined in their own .cs source files. The declaration for a class is:

```
access-specifier class className
{
    // variables

    // methods
}
```

A basic ship class would be like:

```
public class SpaceShip
{
    private float positionX;
    private float positionY;
    private float speed;

    public IncreaseSpeed()
    {
        speed += 10.0f;
    }
}
```

Again, this only defines a template for an object, but doesn't create one. To create an instance of the SpaceShip class we declare a variable of the SpaceShip type and use the new keyword to create it. Then we can access members with the public keyword in front of them just like we do with a struct.

```
SpaceShip ship;
ship = new SpaceShip();
ship.IncreaseSpeed();
```

Going back to the class definition, the first question when looking at this might be what all of those public and private words are. One of the major concepts of OO programming is encapsulation, grouping together data and methods (creating objects.) One of the issues with encapsulation is who can access the data. When we have a variable like the ship's speed we probably don't want other objects going in and changing it. If we had a player that controlled the ship it would make sense to allow the player to tell the ship to increase in speed (call the IncreaseSpeed method) but it wouldn't be good to allow the player class to just set the speed to some arbitrary values, like change it from 0 to 25 (especially if we have physics setup.) To set who can modify what with our class we set access specifiers. For the class and every field and method we give it the specifier of private, public, or protected. Private means nothing outside of the class can do anything with it. With the ship's speed being private no one but the ship can access its speed variable. Public says that anyone else can access it. The ship's IncreaseSpeed method is public, which means any other object can tell the ship to increase its speed. (We'll look at what protected means in moment.) Usually we have all of our data for a class be private and all of its methods be public. This way each class manages its own data (state), but can take messages (have methods called) by other objects.

In programming terms, objects can be thought of as a group of data and actions. This makes sense, as a common way to think about classifying objects is to think about they're characteristics and what they do. For instance, we classify a Ferrari as different from other cars because of attributes it has (different engine, body) and that it can perform actions differently than other cars (go faster.) The data and functions (actions)

are called the members of the object, and the variables are called member variables or properties. The functions contained in the object are called methods of the objects, though a lot of times people will refer to them as member functions. All of these members, variables and methods, are defined in classes.

Where the work is done

When I first learned about OO programming the most confusing wasn't defining classes or setting them up, but figuring out where the work is actually done. We can set up a bunch of objects and methods but how are these used. In most OO programs the main function does almost nothing but create a single object and start it (such as in a windows form application the main function just creates a form class.) So it can be hard to tell how the code is executed.

We can tell how code runs by the arrangement of objects. A common method for organizing programs is to have one (or only a few) main class that everything else is contained in. In XNA we have a Game class. This class holds all of the objects for our game. Inside the main class we create objects like game objects (players, enemies, and so fourth) and objects to handle playing our game, such as a keyboard object to read in key input and graphics device object to handle drawing onscreen. After the objects are created in the game class they run and are updated in the main game loop. To tell how a game is being run we can look in it's main loop and see all of the main objects and how they are being used.

Note that this business of how to "arrange" objects in our programs is pretty controversial. A lot of programmers want to make big objects that contain a lot of stuff so that they can have the object do a great variety of things. But when objects grow too huge, like having hundreds of members and thousands of lines of code they can be confusing and hard to work with. On the other hand, writing a program with hundreds of small objects in it can make things just as difficult. There is a huge variety and debate on how to organize objects in programs.

One thing to watch out for is the tendency a lot of amateur developers have to make wrappers for other objects. Many people want to make helper classes that aren't really needed. For example, in XNA there is a class Texture2D to handle loading pictures for sprites. Some might want to create a Picture class that will be the same as the Texture2D; it will have a Texture2D member and every method of the Picture class will just call the method from Texture2D. Making these wrapper classes is unnecessary and makes code more confusing. (There are special reasons for sometimes writing wrapper classes, like when making an advanced 3D engine, but most people who make them do so when they're not needed).

Inheritance

Getting back to our space ship class, let's say in our program we'll actually have two kinds of ships, a cruiser ship and a fighter ship. They will have many unique differences, like the fighter ship will have a laser cannon but the cruiser won't. But they will also have a lot of common functionality. Instead of repeating a lot of duplicate code to make two totally separate ship classes we can make a single base class called SpaceShip. Then we can make two more ship classes, CruserShip and FighterShip that both inherit from the SpaceShip class. This means the cruiser and fighter classes have everything that the SpaceShip class does but with some extra features too. The code for the fighter ship could look like:

```
public class SpaceShip
{
    private float positionX = 0.0f;
    private float positionY = 0.0f;
    private float speed = 100.0f;

    public IncreaseSpeed()
    {
        speed += 10.0f;
    }
}

public class FighterShip : SpaceShip
{
```

```
    private int ammo = 20;

    public FireCannon()
    {
        ammo--;
    }
}
```

The parent class that we are inheriting from is called the base class. The syntax for inheriting from a class is:

```
public class NewClass : BaseClass
```

Then we can make a FighterShip that has all the functionality of the SpaceShip too:

```
FighterShip ship = new FighterShip();
ship.FireCannon();
ship.IncreaseSpeed();
```

This is the basics on how to inherit from a class. One more note: if we have a member of a class that we want to be private except for the classes that inherit from it, we give it the protected access specifier.

Creating Classes

The previous sections have been a lot of theory, let's get to writing some code. Starting in the second part of this book, we'll begin to make a little space ship fighting game, where the player controls a ship and fights off enemies that fly by. The making of the player would seem to be the first thing we should create, and it makes sense to do so, but here why don't we start making a class to handle enemy ships that fly around, called enemyShip. The first thing we need to think about is what we want the enemies to be able to do. Obviously they must be able to fly, so let's give each enemyShip the methods to moveForword(), moveLeft(), moveRight() to change their position. And this is a 2D game, so we'll record each of their positions with an x coordinate and a y coordinate. We'll also need to handle the player shooting at them and we want them

to be able to be destroyed, so let's give each an integer of hitPoints and a method to handle firedAt(). We'll add more to our enemy class, but for now a basic outline of our basic enemy class is:

```
class enemyShip
float x, y;
int hitPoints;

void setLocation( float xLocation. float yLocation );
void moveForward();
void moveLeft();
void moveRight();
void firedAt();
```

So that's the basic outline, let's create a real class.

Goto File->New Project and create a Console Application with the name enemyShipTest. Then right click on the name enemyShipTest in the Solution Explorer window on the right side of the program. From the right click menu select Add->New Item. A dialog box will appear, make sure Class is selected and in the class name type enemyShip and click Add. Notice in the Solution Explorer window you can click on Program.cs or enemyShip.cs to choose. Add the following to enemyShip.cs to have the file look like this:

```
using System;
using System.Collections.Generic;
using System.Text;

namespace enemyShipTest
{
    class enemyShip
    {
        private float x, y;
        private int hitPoints;

        EnemyShip( int hitPointStart )
         {

        }
        public void setLocation(float xLocation, float yLocation)
```

```
        {

      }

      public void moveForward()
      {

      }

      public void moveLeft()
      {

      }

      public void moveRight()
      {

      }

      public void firedAt()
      {

      }

     public string GetInfoString()
     {
                    return "Ship has location " + x.ToString() + " " + y.ToString()
                    + " and has " + hitpoints.ToString() + " hitpoints"'
     }
  }
}
```

We're going to go through this class line by line and set it up.

The first two lines store variables for the ship.

```
private float x, y;
private int hitPoints;
```

We don't know where the ship will start at, but we should have each ship start with the same number of hitpoints. So change the hitpoints line to:

```
private int hitPoints = 5;
```

When we declare a variable in a class in C# we can go ahead and give it a default value, which is what we just did.

We can also assign default values by passing them to the class when it is created. This is done by a special method called the constructor of the class. The method EnemyShip(int hitPointStart) is the constructor for the EnemyShip class. Change it to look like:

```
EnemyShip( int hitPointStart )
{
    hitPoints = hitPointStart;
}
```

We can tell it is the constructor because the method's name is the same as the class name and it doesn't need any access specifiers in front of it. The constructor is the method called when creating the class:

```
EnemyShip ship = new EnemyShip( 25 ); // Creates a ship with 25 hitpoints
EnemyShip ship2 = new EnemyShip(); // This will cause an error, since we
need to pass the //constructor a number of hitpoints.
```

Going down in code, the setLocation method will just let us set an x and y value for the ship, so all we'll do here is add:

```
public void setLocation(float xLocation, float yLocation)
{
    x = xLocation;
    y = yLocation;
}
```

Now that method is set, it just lets the user specify the location of the ship.

Our next function, moveForward(), will move the player forward in the y direction, so we'll increment the ship's y value.

```
public void moveForward()
{
    y++;
}
```

Similarly, for the moveLeft and moveRight functions we'll just have them change the x value for the ship:

```
public void moveLeft()
{
    x--;
}

public void moveRight()
{
    x++;
}
```

The choice to change the coordinate by 1 is purely arbitrary, we could have moved by more. Now the last method in the class is firedAt(), which when the ship is firedAt() (and we assume it is hit) we'll just decrease the hitPoints by one.

```
public void firedAt()
{
    hitPoints--;
}
```

It may not be fancy, but now we have a working class of an enemy ship. Now let's put it to work.

Creating an Instance

In our main function we'll just create two ships, move them around, and then print out their info:

```
static void Main(string[] args)
{
    EnemyShip ship1 = new EnemyShip(30);
    EnemyShip ship2 = new EnemyShip(35);

    ship1.SetLocation(10,10);
    ship1.MoveLeft();
    ship2.Fire();

    ship1.PrintInfo();
    ship2.PrintInfo();
}
```

The output will be:

```
Ship has location 10 10 and has 30 hitpoints
Ship has location 0 0 and has 34 hitpoints
```

Summary

In this chapter we looked at the basics of Object-Oriented Programming. This just scratched the surface of what OO is and what we can do with it. OO programming is a huge field; check the appendix for some resources to learn more about it. But we've learned enough to start making games: we know the basics of making classes and using them.

Part II: Creating 2D Games with XNA

In the first part of the book we became familiar with the basics of game programming and C# and made some little text based programs. That's all good, but now it's time for things to get more interesting. In this second part we're going to start programming 2D graphics and make a complete game. The game we'll be making is called *Ship Fighter*, it's going to be a 2D top down game, similar to the old *Galaga* game. If you want to see the final result for this section go ahead and download the ShipFighter2D project from xnagamemaking.com and build and run it. In the course of making this game will learn all things you'll need to know to make 2D games with XNA.

Chapter 6: 2D Games and XNA

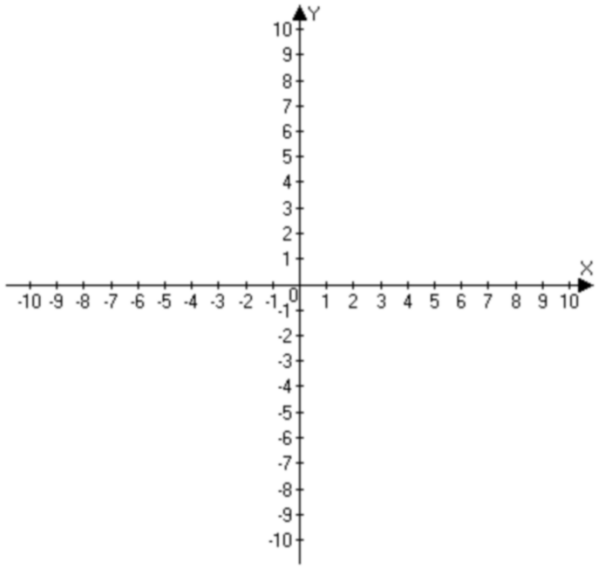
In this chapter we'll start to make our first XNA programs, which will just display pictures on our screen. But before this we'll cover a little bit of theory. We'll keep things short, but we'll go over a bit of 2D math including XNA's 2D coordinate system and vectors. Then we'll take a look at some important game programming concepts and at how sprites work in XNA. Specifically we'll cover:

- How the coordinate system in 2D in XNA works
- Learn the basic properties of sprites
- Create a "Hello World!" type of XNA program
- Draw a sprite onscreen in XNA

2D Graphics Theory

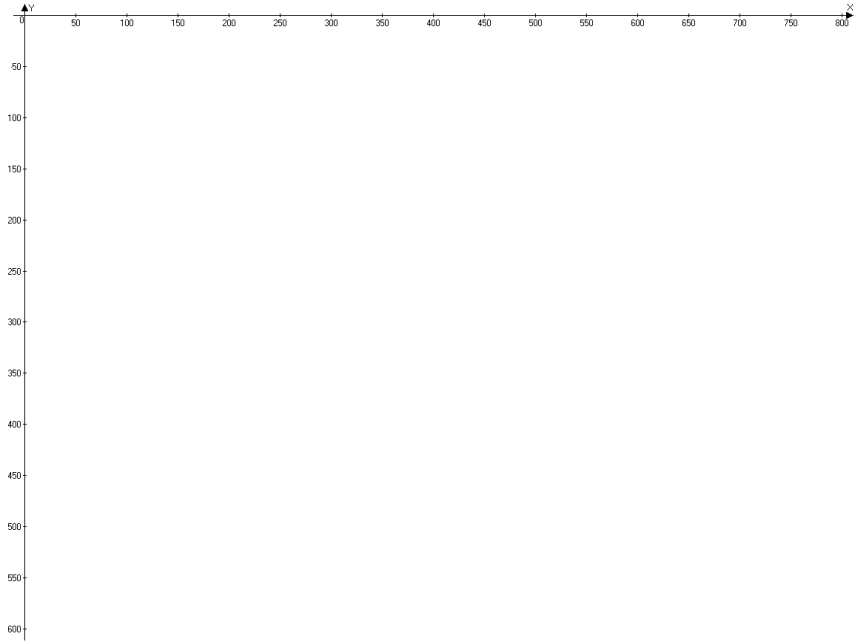
Coordinate System

The most basic thing to do when creating 2D games is to draw various pictures on the screen. To do this we need a way of setting locations for pictures on the screen. If we want to put a picture of a spaceship in the lower center of our screen, how do we say where that is precisely to the computer? You probably remember from school learning about the Cartesian coordinate system. In this system we have two coordinate lines, axises, one that runs vertically called the y axis and another that runs across is the x axis. It looks like this:



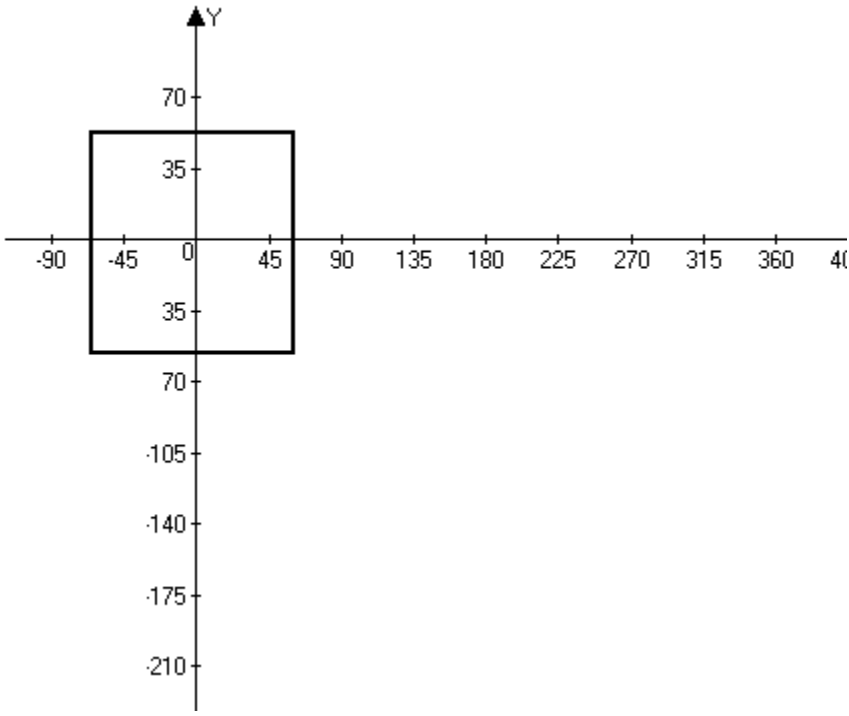
We specify a point, which is just a location, by writing its x and y coordinate, (x,y). The point where the x and y axises cross is called the origin. Points can be negative or positive numbers, depending on where they are at in the grid.

To draw our 2D graphics on the screen we're going to use a similar system; we'll specify locations of all of our graphics using a Cartesian grid with units (units are how for apart each number is) of pixels on the screen. But there is one big difference between our graphics grid and a regular one: the coordinate system for our screen will have the origin in the upper left corner and the positive x-axis going to the right (which is usual) but the positive y-axis starts in the upper left corner and goes down the screen:



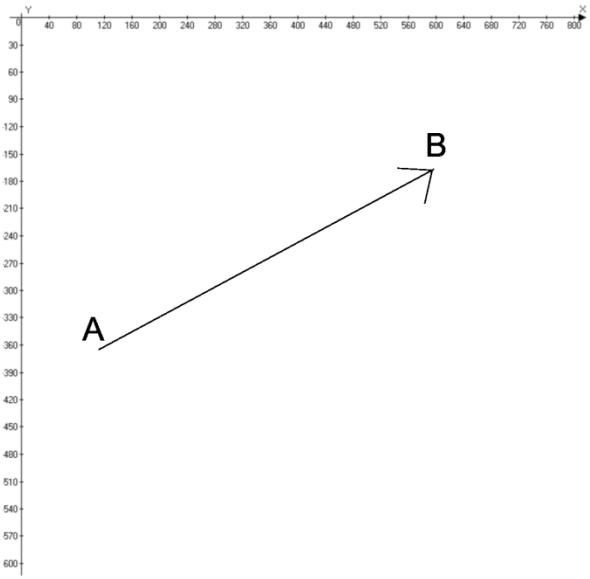
We can think of this grid as just a vertical "flip" of the regular coordinate system. So the point (10,10) is 10 pixels from the left side of the window and 10 pixels from the top. Using this kind of reversed coordinate system isn't anything special about XNA, almost all 2D graphics system use it. At first it can be a little unusual, but it soon becomes natural.

And even though the origin for our coordinate system is in the upper left corner of the screen, there are still negative axises, they're just off screen and can't be seen. For example, let's say that we have a picture that is 100 pixels by 100 pixels and we want to have half of the picture in the top corner of the screen it the picture would be at -50, -50 to 50, 50:

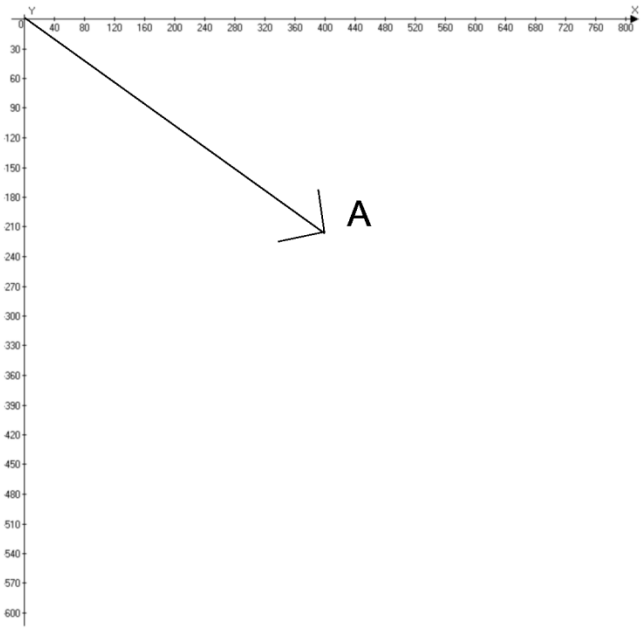


Vectors

Now that we have our coordinate system set up, we need to talk about points and how they are set. When programming with sprites in XNA we technically won't be using points to describe their location, we'll be using vectors. Vectors are mathematical objects that are similar to points except that they don't just describe a location but they also have a magnitude (distance.) We can think of them as a ray from one point to another:



The illustration above shows the vector AB. In most of our 2D programming we'll be using vectors where the first point is at the origin:



You'll notice that this vector is used in a way similar to a point. All it is doing is describing a location (in this section we'll use the term vectors to really mean points.) So why use vectors? The main reason is that vectors are really much more powerful than points. We can do a lot more things with them, especially in 3D. And since the vectors we use to describe location are going to start at the origin and they'll have the same coordinates as points there is no need for a distinct point class. XNA could have created a separate point class to use instead of vectors, but most of the code would be redundant and it would get unnecessarily messy scattered with points and vectors.

How we use vectors in XNA is pretty simple. XNA has objects built in for vectors in two, three, or four dimensions (for some 3D math transformations 4D vectors are needed.) These vectors carry floats for each component. To declare a vector variable we use the Vector keyword followed by its dimension:

```
Vector2 playerPosition;
```

To initialize it:

```
playerPosition = new Vector2( 50.0f, 50.0f );
```

We can set and access its x and y:

```
playerPosition.X = 75.0f;
playerPosition.Y = 75.0f;
```

And we can perform many standard math functions on vectors (adding, subtracting, multiplying, ...)

There are also some defined vectors for standard positions, such as Vector2.Zero for a (0.0f, 0.0f) vector.

Game Programming 101

We just looked a little about the geometry of 2D games, let's look a bit at some game programming concepts.

Game Loop

Video games are a form of interactive computer graphics, meaning that the graphics can be changed in response to input (for instance moving a player around with a controller changes what graphics are drawn on screen.) Of course not all computer graphics are interactive, such as graphics for special effects for movies. Special effects can take a long time to render, maybe hours for a single frame. Graphics for video games must be rendered much quicker, at least 30 frames per second (preferably 60). That's why graphics for movies look so much better than graphics in video games. In movies we can take as long as we like to draw a frame, but a frame for a video game only has one sixtieth of a second to draw everything, and that's not counting time for other non-graphics things such as physics or AI. Since video games are constantly updating (for the interaction) they are designed based on giant loop, a loop that constantly updates everything in the game. The loop can look like the following:

```
while ( Game is Running )
    Check for input (keyboard, mouse, controller)
    Update Entities in the game (player, enemy)
    Update Physics
    Update Collision Detection
    Draw Scene
end while
```

This loop only has a few things in it, a real game loop includes all kinds of items, such as checking for network messages or collision detection. The overall speed of the game loop is measured in the number of frames that are rendered each second (frames per second or FPS). This is also referred as Hertz, Hz. A game working at 30 frames per second could be said to be running at 30Hz. The default speed for XNA games is 60Hz,

but we can change this if we want to (or it can change if we don't want to, such as if someone with an older computer tries to run our game and they can't update that fast.) We'll see the loops we'll use in our game in our first XNA program later in this chapter.

Finite State Machines

Another concept important to game programming is the idea of describing things in terms of finite state machines. A finite state machine (FSM) is just a fancy term for describing things in terms of various "states" they can be in. For instance, let's say we're making a first person shooter game and we have an enemy class for creatures we encounter. An enemy creature could be walking around, or attacking us, or dead and just laying there. We can describe this enemy as a FSM, with states of walking, attacking, or dead. When we go to update the entity in the game we'll change it based on the state the enemy is in. (Like if the enemy is in the walking state we'll have it move, but if it's in the dead state it won't.)

Thinking of things in terms of FSMs makes the entire game design easier. Even a game itself can be thought of as a state machine; the typical game has states such as PLAYING, PAUSED, GAME_OVER, SPLASH_SCREEN. Sometimes when programming these state machines the states will be very explicit. When updating the enemy creature we could have a case statement or series of if's to see which state the enemy is in:

```
if( state == WALKING )
    // walking code
else if( state == ATTACKING )
    // attack player code
else if( state == DEAD )
    // Do nothing
```

and somewhere else we have a function that chooses the value of the state variable for the entity at each update. Or sometimes we keep the conceptual idea of the various states, but in the code it is not spelled out so much. The above code could look like:

```
if( enemyHealth == 0 )
    return; // If destroyed, exit

if( playerDistance < 10 )
    // Attack code
else
    // walking code
```

In this case we use the idea of states, but not explicitly. When creating a design for your games be sure to think of the different objects and the different states they can be in.

Graphics VS. Logic

I come from a background of programming graphics, so it's easy for me to think of game programming as a special case of creating graphics. But most of the work that goes into programming a game is not in its graphics. There are huge areas of programming games that have nothing to do with graphics, such as play control, physics, and artificial intelligence, to just name a few. But graphics are a major portion of any video game, and they tend to use a lot of resources. One of the things we do in game programming is to make a separation of programming logic, which isn't graphics, and the programming (rendering) of the graphics themselves. XNA even has two game loops, one for just updating the graphics and another for everything else. The graphics loop is:

```
protected override void Draw(GameTime gameTime)
```

Which will just have code for rendering everything, and the update loop is:

```
protected override void Update(GameTime gameTime)
```

which will contain logical updates, such as checking input. We'll talk more about these in a moment. Note the two loops run asynchronously,

which means that it is not the case that the update loop updates, then the drawing loop updates, then the update again. Instead they run separately, both running independently (but still at roughly the same rate.)

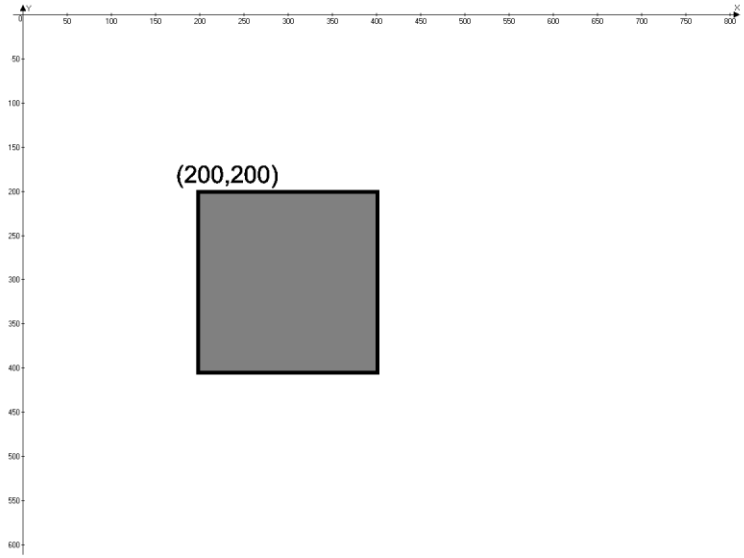
Sprites

We've covered some math and game concepts, let's look at sprites. The 2D game we'll make in this book is a sprite based game. Sprites are bitmap images, just regular pictures, that we can move and do different effects on screen with. (Note the term sprites often mean a series of multiple pictures, like a little sequence of animation, but in XNA the term means a static image.) Sprites are raster images, like photographs, which means that each pixel of the image is stored in memory. Not all 2D games use sprites for their pictures, some use vector graphics. Vector graphics store programmed painting commands to draw the picture and so individual pixels of the image aren't stored in memory. Traditionally 2D games have been made with sprites, but some platforms like Macromedia's Flash use the programmed vector graphics. Sprites give much more detail and more realistic images, and we won't be using vector images in this book. All of the older scroller games from platforms such as the older Segas and Nintendos were made with sprites. We can do a lot with sprites, and we'll cover some basic concepts of them here.

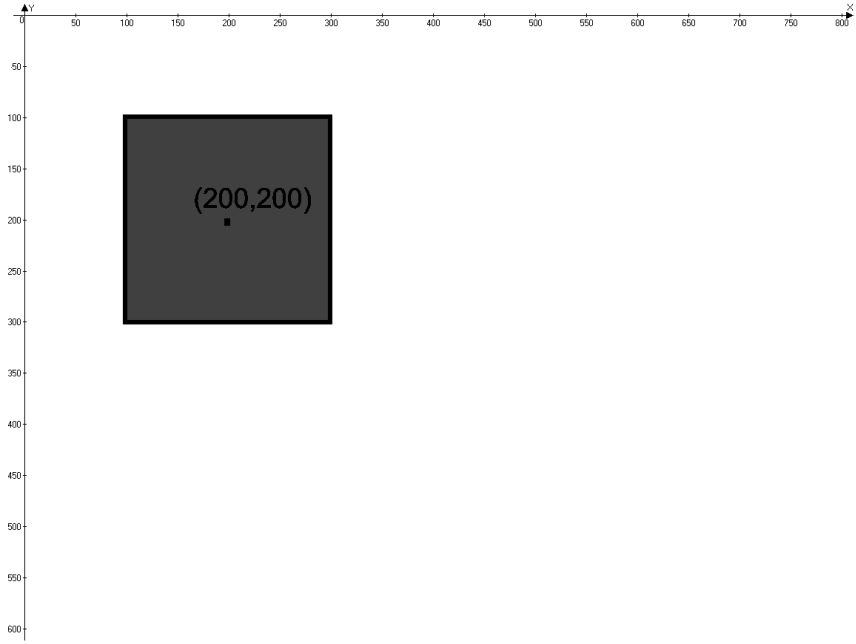
Before we get into sprites here are two notes. First in XNA the documentation calls our pictures sprites, but in the code itself they are called textures (specifically Texture2D objects) which are standard picture files used in graphics. Also note the dimensions of a sprite are just its size measured in pixels. We say this as width by height (and write it as width x height). So a sprite 300 pixels wide and 200 pixels high is a 300 by 200, or 300 x 200 sprite.

Sprite Positioning

We'll position sprites in our game using the coordinate system described before. One thing we need to think about when positioning sprites on the screen is the sprite origin, which is just a specific location on the sprite, that when we tell XNA to draw it puts the sprite there. If we have a sprite that is 200 x 200 and it has a sprite origin of (0,0) if we tell XNA to draw it at point (200.200) it will look like:



But that may not be what we want. If we have a player picture and we want to say "draw the player at position (200,200)" we probably want the player picture to be centered at point (200,200), not have the upper left corner there. Since our picture is 200 x 200, if we put the sprite origin at (100, 100), the center of it, the sprite will be drawn like this:



Which makes sense that the location we specify is at the center. By default the sprite origin is the upper left corner of the sprite, but often we'll set it to the center of a sprite. The sprite origin is just something to keep in mind when drawing sprites, so we don't get unexpected results.

Rotation, Translation, and Scale

The three main things we can do to a sprite in our game our translate it, rotate it, or scale it. Translation means to move the sprite to a different position, rotation means to put a "spin" on it, and scaling a sprite is just shrinking or stretching it. These basic operations, translate, rotate, and scale, are the three basic things we can do to all graphical objects. This is true for 3D graphical objects as well. Usually these operations are done using matrices, but we'll translate rotate and scale using prebuilt functions. And to keep things even simpler we'll assume that the sprites that we use for our games will be made in the size we want already, so we won't be scaling the sprites in our games.

Sprite Alpha and Tinting

All of the pictures that we use for our sprites will be bitmap (.bmp), targa (.tga), .jpeg or .png files. These files are always rectangular, but obviously we want to draw pictures on the screen that aren't rectangles. If we're drawing a spaceship on screen we want the picture to be the shape of the ship, not a rectangle. We can assign parts of the picture an alpha value, a level of transparency in the image. The alpha parts won't be drawn, or can be drawn as partially transparent. This lets us render sprites of different shapes. Also, besides giving the sprites an alpha value we can also tint the sprite, which means giving it an extra overlay color. If we don't want to tint the sprite we can just "tint" it white.

Sprite Depth

When we render sprites on the screen the order in which they are drawn is important. If we have a large background sprite and a smaller spaceship sprite that supposed to go on top of it we would be in trouble if the spaceship sprite was drawn first and the background sprite was drawn on top of it (we wouldn't be able to see the ship.) XNA by default uses what is called a painter's algorithm, where the first things listed are drawn first and later drawing commands draw things on top of the earlier one. We can keep track of what sprites should go on top of each other ourselves; we can just give each sprite a depth. The sprite depth is a decimal number between zero and one. With sprites closer to zero being drawn last (on top of other sprites) and sprites closer to one being drawn first (on the "back" of the screen.) For our spaceship sprite we can give it a depth of zero and our background a depth of one and everything will be drawn correctly.

Sprite Batching

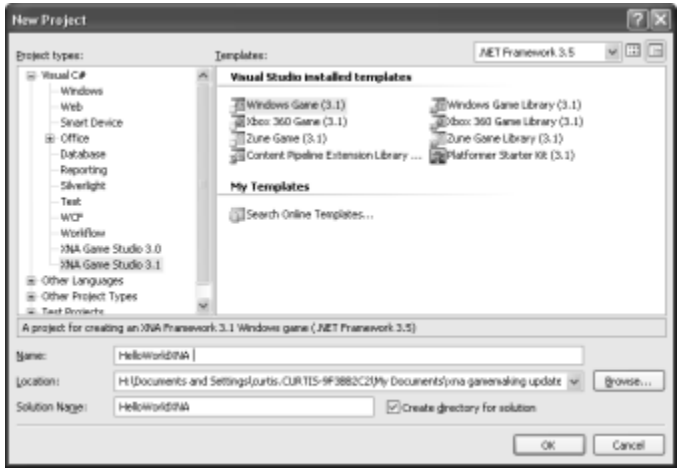
When we render sprites on screen, we don't just tell the computer to render them one at a time. Instead what we have to do is put sprites in groups called sprite batches. Then to render sprites we render the batch of them. There are a lot of good reasons for drawing sprites this way. It can make the process of drawing them faster since they all are rendered at the same time, and easier to program since we can specify options for how they are to be drawn to whole groups of sprites instead of one at a

time. We can also use multiple batches to group sprites together logically. For example, later on we'll be using one batch for the background, a second for the main game elements (player, enemy) and a third for any HUD text.

Hello World! XNA

Finally, we are going to start some programming. Just like we did with C# we'll make an XNA version of Hello World!, except we won't have it print out anything on the screen and we won't add any code to do anything; the only thing it will do is show the default blank screen. Even though we'll just be using the default XNA code we'll go through it line by line and see what's going on.

To create an XNA program start XNA Game Studio and Goto file->New Project. Then select Windows Game (3.1) and type the name HelloWorldXNA for it.



This will create an XNA project and some code will be automatically generated for it. The following code (minus some comments) is here:

```
using System;
using System.Collections.Generic;
using System.Linq;
```

```
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Audio;
using Microsoft.Xna.Framework.Content;
using Microsoft.Xna.Framework.GamerServices;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;
using Microsoft.Xna.Framework.Media;
using Microsoft.Xna.Framework.Net;
using Microsoft.Xna.Framework.Storage;

namespace HelloWorldXNA
{

    public class Game1 : Microsoft.Xna.Framework.Game
    {
        GraphicsDeviceManager graphics;
        SpriteBatch spriteBatch;

        public Game1()
        {
            graphics = new GraphicsDeviceManager(this);
            Content.RootDirectory = "Content";
        }

        protected override void Initialize()
        {
            // TODO: Add your initialization logic here

            base.Initialize();
        }

        protected override void LoadContent()
        {
            // Create a new SpriteBatch, which can be used to draw textures.
            spriteBatch = new SpriteBatch(GraphicsDevice);

            // TODO: use this.Content to load your game content here
        }

        protected override void UnloadContent()
        {
            // TODO: Unload any non ContentManager content here
        }

        protected override void Update(GameTime gameTime)
        {
```

```
            // Allows the game to exit
             if (GamePad.GetState(PlayerIndex.One).Buttons.Back ==
             ButtonState.Pressed)
                this.Exit();

            // TODO: Add your update logic here

            base.Update(gameTime);
        }

        protected override void Draw(GameTime gameTime)
        {
            GraphicsDevice.Clear(Color.CornflowerBlue);

            // TODO: Add your drawing code here

            base.Draw(gameTime);
        }
    }
}
```

If you Goto Debugging->Start Debugging or hit F5 you'll see the following screen appear:

This may not be much, but if you tried writing C++/DirectX code to just put a window on the screen like this you might have to write up to a 100 lines. Now let's analyze the code. Notice in the first part besides the regular .Net C# using directives we now have using statements for the XNA framework as well.

```
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Audio;
using Microsoft.Xna.Framework.Content;
using Microsoft.Xna.Framework.GamerServices;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;
using Microsoft.Xna.Framework.Media;
using Microsoft.Xna.Framework.Net;
using Microsoft.Xna.Framework.Storage;
```

These are all the using directives we need to have access to the complete XNA framework. You could go through these and see what each one is in the XNA documentation. Remember that in any file where you need to use any XNA classes copy these using directives.

The game itself is in the relevantly named Game class:

```
public class Game1 : Microsoft.Xna.Framework.Game
```

Which is a basic framework for a game. All of the games we make with XNA will be inside this class. Notice the code doesn't have a main function like all of our Console programs in the previous part of the book. That's because the Game class takes care of starting and running our program. The game class starts with two data members:

```
GraphicsDeviceManager graphics;
SpriteBatch spriteBatch;
```

The graphics is the graphics device manager and is a very important object. It handles the underlying graphics device that actually draws (renders) everything on screen. We'll modify the graphics device when we want to change the way things are displayed, such as switching our game from windowed mode to full screen. The other member is a SpriteBatch, which as is a batch used for drawing groups of sprites.

After these declarations we find our game class constructor, Game1(). The game constructor just initializes the graphics device manager and sets the name of the directory that all of our game content (sprites, sounds, etc) will be found in:

```
public Game1()
{
    graphics = new GraphicsDeviceManager(this);
    Content.RootDirectory = "Content";
}
```

After this is an initialization method:

```
protected override void Initialize();
```

which if we need to do some things (like calculations) before our game officially starts up we can put them there. We can also load any non graphics content there. The next method:

```
protected override void LoadContent()
```

is a place to load any graphics resources before the game starts. This will be a very important method; eventually we'll be adding a lot of graphics content and it will all be done here. The following method

```
protected override void UnloadContent()
```

Lets us unload any graphics content when the game exits to release it out of memory. We won't be modifying this method.

After this we have two more methods:

```
protected override void Update(GameTime gameTime)
protected override void Draw(GameTime gameTime)
```

These are the game's two main update loops we talked about before. These two loops will be running all the time. Just to go over again, the Update loop is where we update all of the game logic and the Draw loop is where we draw everything on the screen. Each of these is passed an instance of the GameTime class, which tells us things about how much time has passed since the last time the loop was run. (This is helpful for many things, like physics calculations.)

By default the update loop has a line to test if the game has exited, and the Draw loop has a graphics command:

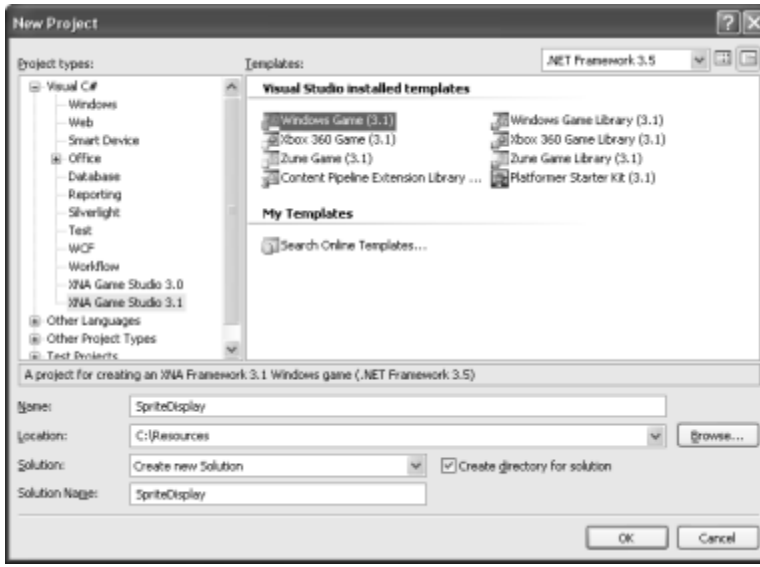
```
GraphicsDevice.Clear(Color.CornflowerBlue);
```

The one graphics line clears the screen to the color Cornflower blue (if you don't like this color you can delete it and type in a different color such as White. To see a list of colors delete the period after Color and write it again and Intellisense, Visual Studio's syntax helper, will display a list of available colors.) The graphics device is cleared at each update of the graphics loop, and then we have one sixtieth of a second to draw everything in our game.

Adding a Sprite

We have created our first XNA program, but it doesn't really do much. So let's get started with sprites and add in a big background sprite, that covers the whole the whole window. To do this we'll add in an object for a Sprite.

Go ahead and make a new project for this program. In C#, goto File -> New Project and select Windows Game (3.1). Name the project SpriteDisplay and note the folder it is being saved in:



The reason to pay attention to the location the project is saved in is because we need to copy our artwork to the content folder there. Get the

spaceBackground.bmp file either from the samples on xnagamemaking.com. Copy the artwork to the content folder inside the folder of the project. For example, using the path C:\Resources above we would copy the picture to C:\Resources\SpriteDislay \SpriteDisplay\Content. With the artwork in the correct location we can then add it to our project.

Right click on Content folder in the solution explorer on the right side of Visual C# and select Add.. -> New Item... Then select spaceBackground.tga and click OK. This adds the sprite to our project.

Now our artwork is setup and onto the sprite code. When we start adding things to our game class in XNA we'll typically do four things:

1. Declare the variable
2. Initialize the variable, usually in the LoadContent() method
3. Add the item to our Draw loop so we can see it on screen
4. Add the item to our Update loop so we can do any logic updates on it

For now we are only going to be creating a sprite (the sprite batch is already created for us). It won't need any update logic so we'll only be doing the first three things.

First we'll declare the variable. Just before the main Game1 class and after the spriteBatch definition add the following line:

```
Texture2D spaceTexture;
```

Now that we have the sprite defined we'll need to initialize it, so change the LoadContent method to

```
protected override void LoadContent()
{
    spriteBatch = new SpriteBatch(GraphicsDevice);
    spaceTexture = Content.Load<Texture2D>("spaceBackground");
}
```

The SpriteBatch initialization is pretty simple; all we do need to do to create it is pass it our graphics device. The syntax

```
spriteBatchName = new SpriteBatch(GraphicsDevice);
```

will be the same for any additional sprite batches we create.

To create the sprite we call the content manager to load it and pass it the name of our texture resource. The content manager is an object that loads all of the art data, not just sprites. The basic syntax for it is:

```
Content.Load<ResourceType>("assetName");
```

Where ResourceType is the type of content we're loading, such as a texture or 3D model, and assetName is the name of our asset. Assets are just the name of the file (without any extension). This syntax will also be used for loading all of our sprites, so the line is worth remembering.

We have the sprite loaded, next let's draw it onscreen. This is a large texture that fills up everything so we'll just position it in the upper left hand corner.

```
protected override void Draw(GameTime gameTime)
{
        GraphicsDevice.Clear(Color.CornflowerBlue);

        spriteBatch.Begin();
        spriteBatch.Draw(spaceTexture, Vector2.Zero, Color.White);
        spriteBatch.End();
        base.Draw(gameTime);
}
```

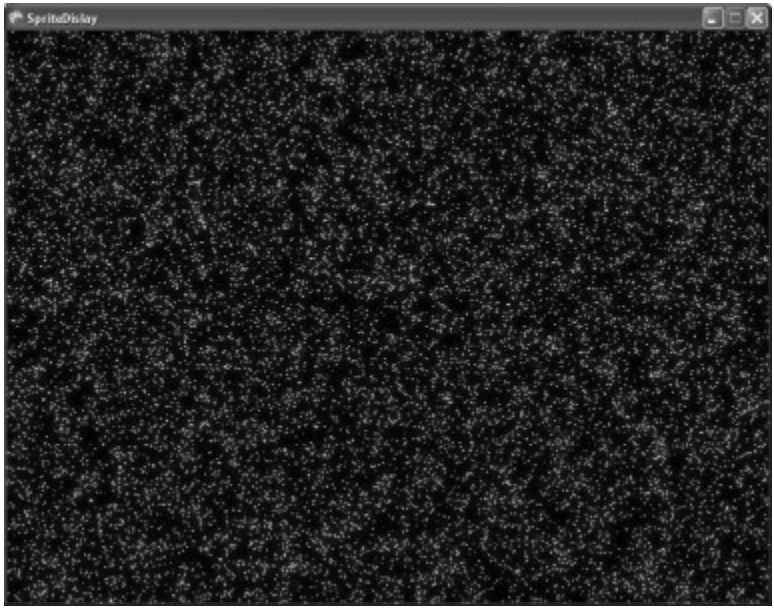
To draw our sprite batches we first start them by calling the method Begin(). After this we call all of our command, to draw the sprites. Then at the end of the SpriteBatch we call .End(), which actually draws everything on the screen.

The spaceBatch.Draw() draws the texture. The syntax for it is:

```
spriteBatch.Draw( spriteName, positionVector, tintColor );
```

The first parameter is the name of the texture to draw. The second parameter is where we want the position of the upper left corner of the sprite to be at. (Again here we put (0,0) for the upper left corner.) The last parameter is for tinting, which we don't want so we'll just set the tint to white.

Go ahead and build/run the program by going to Debug->Start Debugging or hitting F5. You'll see the sprite fill up the screen:



And now we have a program that draws a sprite on the screen. Try changing the Vector2.Zero in the draw method to change the position of the sprite. In the next chapter we'll talk about how to put the other sprite attributes we mentioned (sprite depth, sprite origin...) in XNA code.

Summary

We have two XNA programs under our belt. In the next chapter we'll start making the *Ship Fighter* game, by creating a separate player class for the game and then take in input from the keyboard and Xbox controller to move the sprite around.

Chapter 7: Creating a Player Class

When I first taught a course on XNA after showing how to draw a sprite on the screen I immediately went on to reading in the keyboard state and moving the sprite around on the screen. Then a few classes later when we started to make larger games we changed the program so that the sprites were no longer being stored in the main game class but in separate classes, such as storing the picture of the player in a player class instead of in the game class. This proved to be very confusing for people, and most wanted to just keep the sprites all in the game class. So now, in order to avoid confusion later, we'll start making a separate class to store our sprite for the player in, which is more Object-Oriented and is the way larger games are made.

The class we'll be making is the basics of the Player class for the Ship Fighter game. This class will be pretty simple: It will draw a sprite for the player ship on the screen and move it around. We'll also look at some drawing options to give more details when we're rendering sprites. We will:

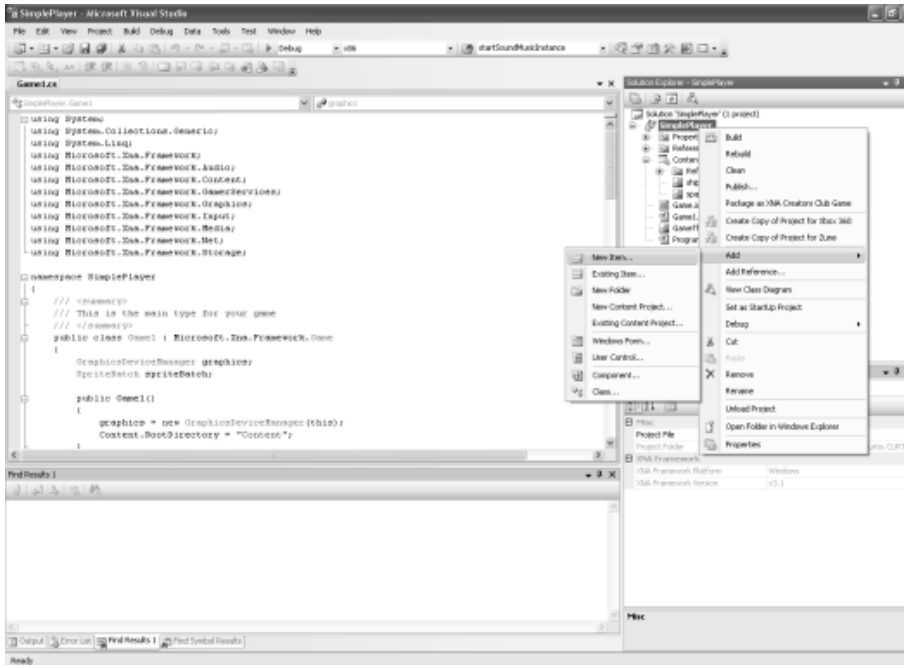
- Create a separate player class for our player sprite.
- Put in more sprite attributes
- Learn how to read input in XNA

Creating a Player Class

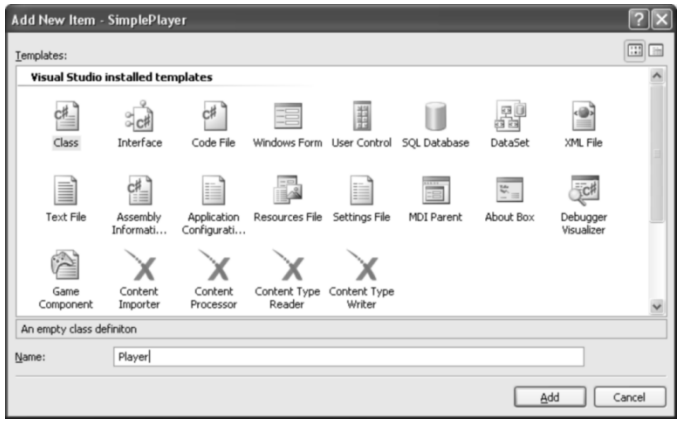
Creating a New Project

We'll start creating the player class for the *Ship Fighter* game. Create a new solution just like we did in the previous chapter by going to File->New Project and select Windows Game (3.1). Name the project SimplePlayer. Get a copy of shipSprite.png and spaceBackground.tga from xnagamemaking.com and copy them to the Content folder of SimplePlayer and add them to the project. (For more details on how to do this look at the creation of the SpriteDisplay solution in the previous chapter. Actually, to keep the background go through the steps in that chapter again to draw the background picture.)

To create the player class the first step is to add a new CS file for the player. Right click on the project name (SimplePlayer) in the solution Explorer and select Add->New Item...



From the Add New Item dialog select Class and give it the name Player and click OK



This will create a holder for the Player class. The following code is automatically generated in Player.cs:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace SimplePlayer
{
   class Player
   {
   }
}
```

The first thing that we'll do is add in the XNA using statements so that we can access XNA code in this file too. Add the following below the System.Text:

```
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Audio;
using Microsoft.Xna.Framework.Content;
using Microsoft.Xna.Framework.GamerServices;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;
using Microsoft.Xna.Framework.Media;
```

```
using Microsoft.Xna.Framework.Net;
using Microsoft.Xna.Framework.Storage;
```

You really don't need all of these using directives, but like we said before we'll just copy all of them to keep things simple. We now have the Player.cs file setup and ready to start adding code.

Player setup

In the last chapter we went over four things to add a variable to the game class in XNA:

1. Declare the variable
2. Initialize the variable, usually in the LoadContent() method
3. Add the item to our Draw loop so we can see it onscreen
4. Add the item to our Update loop so we can do any logic updates on it

For our player we'll be following this pattern. First we'll create the member for the player in Game.cs, then initialize the player, and then add drawing and updating to the Player. The two files we'll be working with here are Player.cs and Game1.cs.

Declare the Player

The first step is easy, in the Game1.cs file or our project before the public Game1() line add the following:

```
Player playerShip;
```

This says we are declaring a new object for our game class, a Player object called playerShip. Note the term Player is because we named our player class Player. The playerShip is the name of the instance of our

Player class, and we can call this anything we want. Our player object is now in our game, but it isn't initialized (loaded) yet.

Initialize the Player

Before initializing the player we need to decide what information we need to track about the player. Our goal for the moment is to just draw a player ship on the screen and move it around, so we'll only need a few member variables for the player. In the player class we'll create variables for the sprite (picture to draw), a Vector2 for the sprite's position onscreen, and the sprite origin. The sprite origin remember is where on the sprite it will center the drawing, and we'll want to keep this in the center of the sprite (this means if we tell the sprite to be drawn at coordinate (200,200) on the screen the center of the sprite will be at (200,200), instead of putting the upper left corner of it there.) In addition to these we'll store the window width and height we're drawing the sprite in, in case we need it later.

So in our player class add the following variables:

```
Vector2 position;
Texture2D shipSprite;
Vector2 spriteOrigin;
int windowWidth, windowHeight;
```

Next let's create a constructor for the Player, which will initialize these variables:

```
// This method is called when the Player is created
public Player(GraphicsDevice device, Vector2 position, Texture2D sprite)
{
    // The position that is passed in is now set to the position above
    this.position = position;

    // Set the Texture2D
    shipSprite = sprite;
```

```
    // Setup origin
    spriteOrigin.X = (float) shipSprite.Width / 2.0f;
    spriteOrigin.Y = (float)shipSprite.Height / 2.0f;

    // Set window dimensions
    windowHeight = device.Viewport.Height;
    windowWidth = device.Viewport.Width;
}
```

This constructor takes in a GraphicsDevice, position, and Texture2D. For the player's position we set the position passed in as our start position. Then for the shipSprite we set it to the sprite passed in. For the origin we just take the center of the shipSprite, which is found by taking its width and height and dividing by two:

```
spriteOrigin.X = (float) shipSprite.Width / 2.0f;
spriteOrigin.Y = (float) shipSprite.Height / 2.0f;
```

Then for the window width and height we look at the graphics device's viewport (drawing area) width and height:

```
windowHeight = device.Viewport.Height;
windowWidth = device.Viewport.Width;
```

That sets up our player constructor. Let's initialize the player back in the game class. Change the LoadContent method in the Game1.cs to this:

```
protected override void LoadContent()
{
    spriteBatch = new SpriteBatch(GraphicsDevice);
    spaceTexture = Content.Load<Texture2D>("spaceBackground");
    Texture2D playerTexture = Content.Load<Texture2D>("shipSprite");
    playerShip = new Player(GraphicsDevice, new Vector2(400, 300),
            playerTexture);
}
```

The first thing new here is the loading of the sprite for the player ship. Note that we declare this as a local variable since we won't be using the

playerTexture in the Game1.cs file any after we create the player. Then we initialize the player by giving it the current GraphicsDevice, a start position Vector2 (which doesn't have to be (400,300), this was just chosen because it puts it in the center), and the sprite for the player ship.

If you try running the program now by going to Debug->Start Debugging you'll find that nothing seems to happen; it still just shows the background from last time but no ship. This is because we have initialized the player but we are not drawing or updating it yet.

Drawing the Player

To draw the player we'll first add a draw method to the player class. Right below the Player constructor method in Player.cs add the following:

```
 // Draw the player
public void Draw(SpriteBatch batch)
{
    batch.Draw(shipSprite, position, null, Color.White,
               0.0f, spriteOrigin, 1.0f, SpriteEffects.None, 0.0f);
}
```

This method takes in a SpriteBatch and then uses it to draw the shipSprite on the screen. As you can see the batch.Draw looks a bit different than the ones we've seen before. We'll go over the changes in a minute, but for now change the Draw method in Game1.cs to:

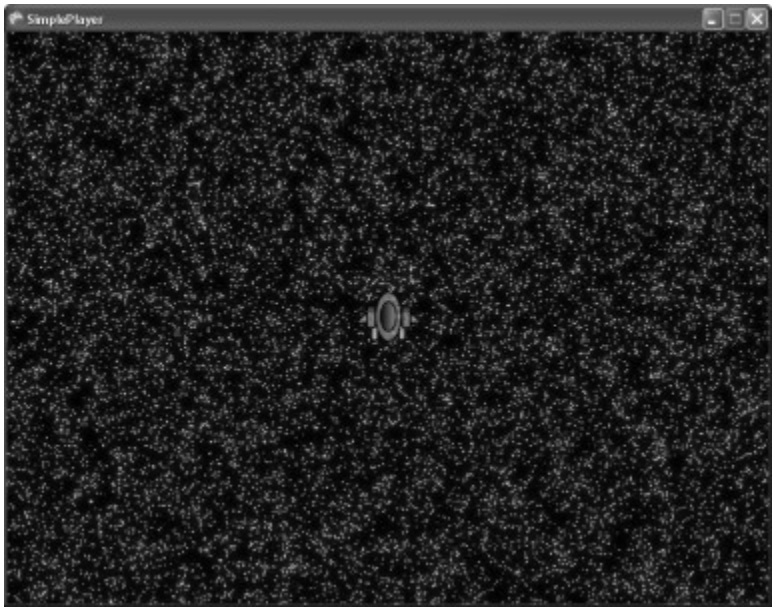
```
protected override void Draw(GameTime gameTime)
{

    GraphicsDevice.Clear(Color.CornflowerBlue);

    spriteBatch.Begin();

    spriteBatch.Draw(spaceTexture, Vector2.Zero, Color.White);

    playerShip.Draw(spriteBatch);
```

```
    spriteBatch.End();

    base.Draw(gameTime);
}
```

The key line here is the:

```
playerShip.Draw(spriteBatch);
```

This calls the player's draw method and tells it to draw the ship sprite on the screen. If you run the program you'll see this:



We're drawing the ship in our game by putting it in the draw loop and adding a draw method to the Player class. Next we'll look at that batch.Draw method in the Player class again.

Sprite Attributes

The draw method we used this time was:

```
batch.Draw(shipSprite, position, null, Color.White,
                0.0f, spriteOrigin, 1.0f, SpriteEffects.None, 0.0f);
```

This matches up with this definition:

```
public void Draw (
        Texture2D spriteName,
        Rectangle destinationRectangle,
        Color tint,
        float rotation,
        Vector2 origin,
        SpriteEffects effects,
        float layerDepth
)
```

The first parameter, spriteName, is the name of the sprite to draw. The second parameter is for a destination rectangle. If we want just part of the image to be drawn we specify a rectangle to be the size of the sprite on screen. If you try replacing the null in our code with the line new Rectangle(0,0,40,100) and run the program you'll see only half of the spaceship, since the destination rectangle is half the size of our original image. This parameter we'll keep at null most of the time.

The third parameter is for adding a tint to our image. The parameter is a color, (if you type Color. a list of colors will be available.) We don't want a tint so we'll just keep the color white. The rotation allows us to set an angle of rotation for the sprite, if we want it a different orientation. The next parameter is a vector for the sprite origin. Remember for a spaceship sprite that we move along the screen we don't think of the spaceship's position as where its upper left corner, so we'll put the origin at the center of the spaceship sprite. This way the position of the spaceship means its center.

The SpriteEffects parameter allows us to flip the picture if we want to (the enumeration is FlipHorizontally and FlipVertically) but, again, we'll leave that on None. The last parameter is the layer depth. The sprite

depth, if you recall, is the order in which the sprites are drawn on top of each other, with those closest to 0.0 being nearer to the top, and those close 1.0 being at the bottom. So we give the spaceship a depth of 0.0 since it's on top and the background a depth of 1.0.

We have the ship drawing, now for moving it around.

Updating the Player

Our goal for the player is to have the sprite moving around based on input. First we'll put in the basic structure for the update. In the player class add the following method below the Draw() one:

```
// Update - for animation
public void Update(GameTime gameTime)
{

}
```

And change the Update loop in the Game1 class to update the player too:

```
protected override void Update(GameTime gameTime)
{
    // Allows the game to exit
    if (GamePad.GetState(PlayerIndex.One).Buttons.Back ==
        ButtonState.Pressed)
    this.Exit();

    playerShip.Update(gameTime);

    base.Update(gameTime);
}
```

We'll be using this a little later. One thing we can do now is creating some methods to change the ship's position. From Game1.cs we'll be getting keyboard and game controller input, and based on that we'll want

to call some functions to move the player sprite around. So add the following four methods to the Player class:

```
 // These match up with the Arrow keys
public void Accelerate()
{
    position.Y -= 3.0f;
}

public void MoveReverse()
{
    position.Y += 3.0f;
}

public void TurnRight()
{
    position.X += 3.0f;
}

public void TurnLeft()
{
    position.X -= 3.0f;
}
```

These methods move the shipSprite around on the screen by changing its position. The first two, Accelerate and MoveReverse, move the shipSprite up and down the screen respectively. Note to move the shipSprite up the screen we subtract from its Y coordinate. Since the zero for Y is at the top of the screen subtracting Y moves up and adding to Y moves down. Turning left and right change the X position. Now we are setup for input.

XNA Input

For input in XNA we have three basic options, the keyboard, Xbox controller, and mouse (we won't worry about the mouse here.) For getting input XNA works a little bit differently than a lot of systems. XNA uses a polling method instead of an event driven method. An event driven method is one where the user does something to generate input, like pressing a key on a keyboard, and then a message is sent to the

program letting it know the input event just happened. XNA's polling method does not generate events like this; what XNA does is at every update of the Update loop you can request the current keyboard state. This state is a snapshot of the keyboard and you can check this snapshot of the keyboard and see what keys are down. This is faster, but we'll need to be careful later when we want to see if buttons are pressed that toggle options. To see how this works copy the following UpdateInput method into the Game class below the Update loop.

```
private void UpdateInput()
{
    KeyboardState keyState = Keyboard.GetState();
    GamePadState gamePadState = GamePad.GetState(PlayerIndex.One);

    if (keyState.IsKeyDown(Keys.Up)
      || gamePadState.DPad.Up == ButtonState.Pressed)
    {
        playerShip.Accelerate();
    }
    if (keyState.IsKeyDown(Keys.Down)
      || gamePadState.DPad.Down == ButtonState.Pressed)
    {
        playerShip.MoveReverse();
    }
    if (keyState.IsKeyDown(Keys.Left)
      || gamePadState.DPad.Left == ButtonState.Pressed)
    {
        playerShip.TurnLeft();
    }
    if (keyState.IsKeyDown(Keys.Right)
      || gamePadState.DPad.Right == ButtonState.Pressed)
    {
        playerShip.TurnRight();
    }
}
```

In the main Game update loop add the line:

```
UpdateInput();
```

This works just like we described. The first two lines get the current state of the keyboard and (optionally) Xbox controller:

```
KeyboardState keyState = Keyboard.GetState();
GamePadState gamePadState = GamePad.GetState(PlayerIndex.One);
```

For the Gamepad (Xbox controller) we can get the state of up to four controllers, so we tell it PlayerIndex.One to get the state of the first one.

After this we just query if a key or gamepad D button is currently pressed. For the keyboard we use

```
keyState.IsKeyDown(Keys. )
```

Which says if the key is down of the argument we pass in. The Keys is an enumeration of all the keyboard keys (type in Keys. and wait for intellisense to show a list of all possible keys.) For the Gamepad we just check the state of the DPad (directional pad) and see if its state is pressed too. We'll be using more Xbox controller buttons later, if you're curious about how to check the rest for now look in the XNA documentation.

If you run the solution now you'll be able to move the ship around on the screen with the keyboard or gamepad. This is quite a big start for our game, having a player class setup and moving a player around onscreen. Next we'll add a few things to it.

Extending the Player

One simple thing we can add to the player class is to keep the player always on screen. Currently you can move the player completely off screen, but it'd be nice to keep it always visible. And right now the ship moves up/down and left/right just at a constant velocity (speed) of 3 pixels per update. It'd be nice to make this be a bit more realistic, having the ship accelerate and decelerate with changing speeds. We'll do some very simple physics to put that in. This solution is just an extension of

SimplePlayer, but is listed as Simple Player Extended on xnagamemaking.com.

Keeping Onscreen

To keep the ship onscreen we'll make use of the window width and height variables that we put in earlier. All we'll do is change the Update method of the Player to the following:

```
public void Update(GameTime gameTime)
{
    if (position.Y < 0) position.Y = 0.0f;
    if (position.Y > windowHeight) position.Y = windowHeight;
    if (position.X < 0) position.X = 0.0f;
    if (position.X > windowWidth) position.X = windowWidth;
}
```

At each update we're checking to see if the X or Y coordinate of the position is below zero, which is off-screen, and if it is just setting the position to zero. And if the X or Y is greater than the window size, which is also off-screen, we max out the position to it. We are locking max and min positions so the ship is always onscreen now.

Putting in Simple Physics

On to putting in acceleration/deceleration. We're not actually putting in real physics here, just some quick hacks to give the ship a physics feel. What we'll do is create a variable called velocity that tracks how fast the ship is going. At each update we'll add to the vertical position the velocity times how much time has padded. And though in space there is no resistance (slow down) we'll go ahead and put in a resistance that shrinks the velocity at each update. This will give us an acceleration and deceleration for the ship.

So first in the Player class add a new variable called velocity below the windowWidth and windowHeight variables:

```
 // Up/Down speed
float velocity = 0.0f;
```

This initializes the velocity to zero, which makes sense since the ship won't start out moving. Then change the Accelerate() and MoveReverse() methods to just change the velocity directly:

```
public void Accelerate()
{
   velocity -= 20.0f;
}

public void MoveReverse()
{
   velocity += 20.0f;
}
```

The choice of 20 is completely arbitrary. The units here are just pixels per second and 20 felt like a good choice. Next we'll put in the position update. This should go right before our window bounds check in the Player update:

```
position.Y += velocity * (float)gameTime.ElapsedGameTime.TotalSeconds;
velocity *= 0.95f;
```

This does what we just said it would. The first line adds to the current position the velocity time the elapsed time. The second line shrinks the velocity by multiplying by 0.95 at each update. Again, the choice of 0.95 is arbitrary; feel free to play with different values for it.

Summary

We have the start of our *Ship Fighter* game setup. We have a player class setup and can move it around onscreen, and saw how to add two simple adjustments to it. In the next chapter we'll put in a multi-level scrolling background for it.

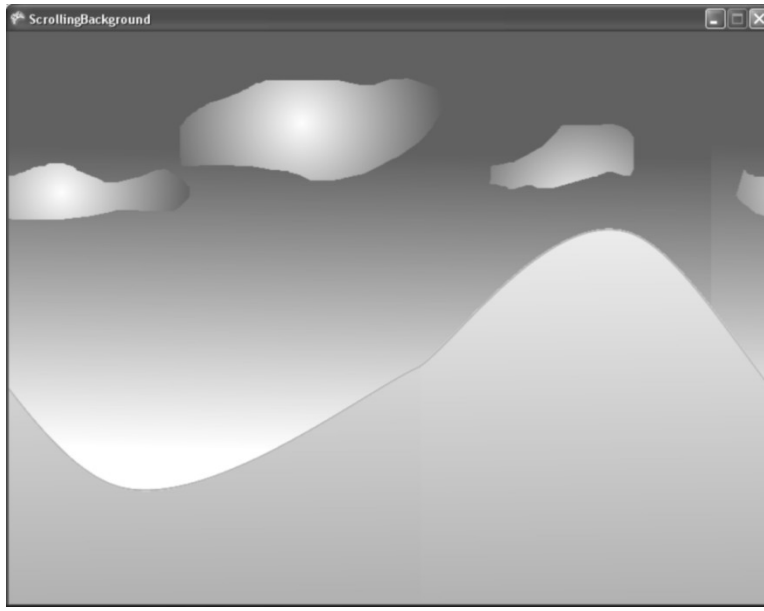
Chapter 8: Scrolling Backgrounds

One of the most popular things to do with 2D games is to have backgrounds that have several layers that move at different speeds. You've probably seen this effect in 2D scrolling games where a character is walking across the screen and the immediate background behind the character is moving at one speed and a background farther in the distance (like a sky far away) is moving at a different speed a little bit slower. This creates a 3D like effect even though the whole game is still 2D. The technical term for these backgrounds with different layers is parallax backgrounds, but here we'll just call them multi-backgrounds or scrolling backgrounds. We'll also talk a bit about including third party code mixed in with your own. Specifically we will:

- Look at a scrolling background class.
- Put the scrolling background into our ship game.

Scrolling Background Example Project

Download and build the Scrolling Background from xnagamemaking.com. If you run this program you'll see a scrolling background with two layers:



If you press the S key it will stop the moving, the M key will restart it. The left and right arrows let you manually move the backgrounds, and the 'U' and 'R' keys will let the background move up and down or left and right. We won't go over the details of the Game1.cs file for this project, we'll just look at the MultiBackground.cs, which contains the code that handles the background. The MultiBackground class works by keeping a list of the different background layers and moving and drawing each based on how they are set. Here is the full listing for the code:

```
class BackgroundLayer
{
    public Texture2D picture;
    public Vector2 position = Vector2.Zero;
    public Vector2 offset = Vector2.Zero;
    public float depth = 0.0f;
    public float moveRate = 0.0f;
    public Vector2 pictureSize = Vector2.Zero;
    public Color color = Color.White;
}
```

```
class MultiBackground
{
    private bool moving = false;
    private bool moveLeftRight = true;

    private Vector2 windowSize;

    private List<BackgroundLayer> layerList = new
      List<BackgroundLayer>();

    private SpriteBatch batch;

    public MultiBackground(GraphicsDeviceManager graphics)
    {
        windowSize.X = graphics.PreferredBackBufferWidth;
        windowSize.Y = graphics.PreferredBackBufferHeight;
        batch = new SpriteBatch(graphics.GraphicsDevice);
    }

    public void AddLayer(Texture2D picture, float depth, float moveRate)
    {
        BackgroundLayer layer = new BackgroundLayer();
        layer.picture = picture;
        layer.depth = depth;
        layer.moveRate = moveRate;
        layer.pictureSize.X = picture.Width;
        layer.pictureSize.Y = picture.Height;

        layerList.Add(layer);
    }

    public int CompareDepth(BackgroundLayer layer1, BackgroundLayer
      layer2)
    {
        if (layer1.depth < layer2.depth)
            return 1;
        if (layer1.depth > layer2.depth)
            return -1;
        if (layer1.depth == layer2.depth)
            return 0;
        return 0;
    }

    public void Move(float rate)
    {
        float moveRate = rate / 60.0f;
```

```
        foreach (BackgroundLayer layer in layerList)
        {
            float moveDistance = layer.moveRate * moveRate;

            if (!moving)
            {
                if (moveLeftRight)
                {
                    layer.position.X += moveDistance;
                    layer.position.X = layer.position.X % layer.pictureSize.X;
                }
                else
                {
                    layer.position.Y += moveDistance;
                    layer.position.Y = layer.position.Y % layer.pictureSize.Y;
                }
            }
        }
    }

    public void Draw()
    {
        layerList.Sort( CompareDepth );

        batch.Begin();

        for (int i = 0; i < layerList.Count; i++)
        {
            if (!moveLeftRight)
            {
                if (layerList[i].position.Y < windowSize.Y)
                {
                    batch.Draw(layerList[i].picture, new Vector2(0.0f,
                        layerList[i].position.Y), layerList[i].color);
                }
                if (layerList[i].position.Y > 0.0f)
                    batch.Draw(layerList[i].picture, new Vector2(0.0f,
                        layerList[i].position.Y - layerList[i].pictureSize.Y),
                        layerList[i].color);
                else
                    batch.Draw(layerList[i].picture, new Vector2(0.0f,
                        layerList[i].position.Y + layerList[i].pictureSize.Y),
                        layerList[i].color);
            }
            else
```

```
            {
                if (layerList[i].position.X < windowSize.X)
                {
                    batch.Draw(layerList[i].picture, new
                        Vector2(layerList[i].position.X, 0.0f ), layerList[i].color);
                }
                if (layerList[i].position.X > 0.0f )
                    batch.Draw(layerList[i].picture, new
                        Vector2(layerList[i].position.X - layerList[i].pictureSize.X,
                        0.0f), layerList[i].color);
                else
                    batch.Draw(layerList[i].picture, new
                        Vector2(layerList[i].position.X + layerList[i].pictureSize.X,
                        0.0f), layerList[i].color);

            }
        }


    batch.End();
}

public void SetMoveUpDown()
{
    moveLeftRight = false;
}

public void SetMoveLeftRight()
{
    moveLeftRight = true;
}

public void Stop()
{
    moving = false;
}

public void StartMoving()
{
    moving = true;
}

public void SetLayerPosition(int layerNumber, Vector2 startPosition)
{
    if (layerNumber < 0 || layerNumber >= layerList.Count) return;
```

```
            layerList[layerNumber].position = startPosition;
        }

        public void SetLayerAlpha(int layerNumber, float percent)
        {
            if (layerNumber < 0 || layerNumber >= layerList.Count) return;

            float alpha = (percent / 100.0f);

            layerList[layerNumber].color = new Color(new Vector4(0.0f, 0.0f, 0.0f,
                alpha));
        }

        public void Update(GameTime gameTime)
        {

            foreach( BackgroundLayer layer in layerList )
            {
                float moveDistance = layer.moveRate / 60.0f;

                if (moving)
                {
                    if (moveLeftRight)
                    {
                        layer.position.X += moveDistance;
                        layer.position.X = layer.position.X % layer.pictureSize.X;
                    }
                    else
                    {
                        layer.position.Y += moveDistance;
                        layer.position.Y = layer.position.Y % layer.pictureSize.Y;
                    }
                }
            }
        }
}
```

The idea when going through this is that you don't need to understand every detail of this file, but just enough to be able to use it. We'll go through a few big points in the code here and then see how to use it in the *Ship Fighter* game.

The first thing to look at is the definition of a background layer:

```
class BackgroundLayer
{
    public Texture2D picture;
    public Vector2 position = Vector2.Zero;
    public Vector2 offset = Vector2.Zero;
    public float depth = 0.0f;
    public float moveRate = 0.0f;
    public Vector2 pictureSize = Vector2.Zero;
    public Color color = Color.White;
}
```

This shows you the different attributes we track abut each layer. We keep a picture of each (of course), its position and any initial start position offset (like if we wanted the layer to start in the middle of the screen). We also keep the sprite depth, which is very important since that decides what order the layers are drawn in. We also keep how fast it is moving (moveRate) which is in pixels per second. And we keep any tint for the layer, which is the color member.

For the ScrollingBackground class, which manages the background layers, we'll only look at a few important methods. The first is the constructor for the ScrollingBackground, that takes in a GraphicsDeviceManager, which it uses to get the current windowSize:

```
public MultiBackground(GraphicsDeviceManager graphics)
```

So to create a ScrollingBackground we initialize it by passing the device manager in:

```
MultiBackground background;
background = new MultiBackground(graphics);
```

To create different layers we use the AddLayer method:

```
public void AddLayer(Texture2D picture, float depth, float moveRate)
```

There are three parameters to this method. The first takes in the sprite to be used for this layer. The second parameter is the sprite depth for this layer, with zero being on top and 1 at the bottom. The last parameter is the moveRate, which is how fast the later will move when the default moving is turned on. An example of using this method is:

```
Texture2D sky = Content.Load<Texture2D>("SkyLayer");
 background.AddLayer(sky, 0.5f, 200.0f);
```

Another important method is Move():

```
public void Move(float rate)
```

This moves everything based on a rate, which is the amount of time for everything to move. So if a layer has a moveRate of 150 pixels per second to move it 75 pixels we would do this:

```
background.Move(0.5f);
```

Another big method is the Draw one:

```
public void Draw()
```

Which is pretty simple to use, we just add to the draw loop:

```
background.Draw();
```

likewise there is a Update method that belongs in the game Update loop:

```
public void Update(GameTime gameTime)
```

Two other methods:

```
public void SetMoveUpDown()
public void SetMoveLeftRight()
```

Just set of the background should move up and down or left and right.

The next two methods stop and start the animation (automatic scrolling):

```
public void Stop()

public void StartMoving()
```

And there are methods to set the default start position for a layer and an alpha tint to a layer.

Adding a Background to the Ship fighter

We've seen how our scrolling background works, let's add our new background to the ship game. We'll only use one layer of it, the same space background it currently has, but will make it scroll down the screen. The source for this project will take up where we left off on SimplePlayer Extended and this project is at xnagamemaking.com called SimplePlayer Extended with Background.

The first thing we'll need to do to our project is add the MultiBackground.cs to it. You can find the MultiBackground.cs in the ScrollingBackground solution or at xnagamemaking.com. Copy MultiBackground.cs to the source folder of the SimplePlayer Extended project (where the Game1.cs file is) and add it to the project by right clicking on the project name in Solution Explorer and choosing Add->Existing Item and select MultiBackground.cs. (This is similar to the way we added the Player.cs file in the last chapter, except here we are not creating a new item but adding an existing one.)

After we have MultiBackground.cs added to the project the first thing we want is to have that code available for the Game class. The namespace of

the MultiBackground.cs is xnaExtras, so in the using statements in the Game1.cs file add the following:

```
using xnaExtras;
```

This allows us to put a MultiBackground object into the Game class. Remember the four basic steps to adding an object to our game:

1. Declare the object
2. Initialize the object, usually in the LoadContent() method
3. Add the item to our Draw loop so we can see it onscreen
4. Add the item to our Update loop so we can do any logic updates on it

So the first thing we need to do is declare the MultiBackground object. To do this we'll delete the line:

```
Texture2D spaceTexture;
```

and replace it with:

```
MultiBackground spaceBackground;
```

since we won't be drawing a sprite for the background but will be using the MultiBackground object instead. Now we have a MultiBackground object declared with the name spaceBackground. Next we need to initialize it, so change the LoadContent method to:

```
protected override void LoadContent()
{
    spriteBatch = new SpriteBatch(GraphicsDevice);

    // This method is the constructor found in MultiBackground.cs
    spaceBackground = new MultiBackground(graphics);
    // Create the texture/sprite for the hill
```

```
    Texture2D spaceTexture =
            Content.Load<Texture2D>("spaceBackground");
    // Add the space layer to the background, this is also in
    // MultiBackground.cs
    spaceBackground.AddLayer(spaceTexture, 0.0f, 200.0f);
    spaceBackground.SetMoveUpDown();
    spaceBackground.StartMoving();

    Texture2D playerTexture = Content.Load<Texture2D>("shipSprite");
    playerShip = new Player(GraphicsDevice, new Vector2(400, 300),
            playerTexture);
}
```

Here to initialize the background first we created it and the spaceTexture for it. Then we added a layer. Remember the three parameters we pass to the AddLayer method is the sprite name, it's sprite depth (zero being closest to the bottom), and default speed (200 pixels per second). After this we set the background to move up and down (instead of the default left to right) and then tell it to start moving. If you run the project now it won't work yet. Next we have to draw it.

To draw the background change the SpriteBatch drawing method in the Game class to draw the background right after clearing the screen:

```
graphics.GraphicsDevice.Clear(Color.CornflowerBlue);

spaceBackground.Draw();
```

We want the background to be the lowest layer on the screen so we'll always draw it first right after clearing the screen. If your run the program you'll see the background on the screen again, but it's not moving. We still need to update it, so add the following to the Game update loop:

```
spaceBackground.Update(gameTime);
```

Now we have a scrolling background behind the ship.

Summary

This chapter has been a little different than the previous ones, in that we just took a source file, looked at the key parts of it, and added it to our ship project. The next chapter will be similar to this in that we'll be adding animation to the project.

Chapter 9: Animation

One of the most important things to do in a 2D game is to create animation. If you have a character that is walking across the screen you don't want just a single from sliding across, but several frames that are animated in a walk sequence. XNA doesn't have sprite animation built in to it (not that it needs to) so we'll be making a new class here that handles the animation for us. Then we'll add the animation to our *Ship Fighter* game. Our goals are:

- Create a class to handle sprite animation
- Put animation in a scroller game.
- Put animation into our ship game.

Changing Screen Size

Before going into animation let's see how to change the screen size for XNA. This doesn't have anything to do with animation, but is good to know. The default screen size for XNA is 800 pixels wide and 600 pixels tall. To change the screen size we change the back buffer width and height of the graphics device. The back buffer is a place in memory where the screen refreshes are originally drawn to, and then when the screen has been refreshed the back buffer is placed onto the monitor. This writing to an off-screen back buffer and then flipping it to the screen buffer is called double buffering. Here is a little function that changes the screen size, that can go anywhere in the Game1.cs file.

```
public void SetWindowSize(int x, int y)
{
    graphics.PreferredBackBufferWidth = x;
    graphics.PreferredBackBufferHeight = y;
    graphics.ApplyChanges();
}
```

To use it we just call it after we initialize the graphics device manager:

```
public Game1()
{
    graphics = new GraphicsDeviceManager(this);
    Content.RootDirectory = "Content";
    SetWindowSize(1280, 1024);
}
```

Remember if you want your game to run in full screen mode smoothly you should set the size to a standard screen size (which can be found by right clicking of the desktop of you computer, selection properties and looking at sizes in the Settings tab.) Otherwise you can pick any setting you like, as long as it is not bigger than you graphics card can handle.

Animation Sample

Back to animation, download and run the Animation sample from xnagamemaking.com. The player that appears onscreen can move left and right with the arrow keys and pressing A will attack. The main file of this project is Animation.cs; the Animation class works by having different cells of animation (sprites) added to it, then you can play or loop through a series of them. Here is the source code for it:

```
struct AnimationCell
{
    public Texture2D cell;
}

class Animation
{
    int currentCell = 0;

    bool looping = false;
    bool stopped = false;
    bool playing = false;
    // Time we need to goto next frame
    float timeShift = 0.0f;

    // Time since last shift
    float totalTime = 0.0f;
    int start = 0, end = 0;
    List<AnimationCell> cellList = new List<AnimationCell>();
```

```
Vector2 position;

float scale = 1.0f;
SpriteEffects spriteEffect = SpriteEffects.None;

public float Scale
{
   set
   {
      scale = value;
   }
}

Vector2 spriteOrigin = Vector2.Zero;

public Vector2 SpriteOrigin
{
   set
   {
      spriteOrigin = value;
   }
}

public Animation(Vector2 position)
{
   this.position = position;
}

public void AddCell( Texture2D cellPicture )
{
   AnimationCell cell = new AnimationCell();
   cell.cell = cellPicture;

   cellList.Add(cell);
}

public void SetPosition(float x, float y)
{
   position.X = x;
   position.Y = y;
}

public void SetPosition(Vector2 position)
{
   this.position = position;
```

```
    }

    public void SetMoveLeft()
    {
        spriteEffect = SpriteEffects.FlipHorizontally;
    }

    public void SetMoveRight()
    {
        spriteEffect = SpriteEffects.None;
    }

    public void LoopAll(float seconds)
    {
        if (playing) return;

        stopped = false;
        if (looping) return;

        looping = true;
        start = 0;
        end = cellList.Count - 1;

        currentCell = start;
        timeShift = seconds / (float)cellList.Count;
    }

    public void Loop(int start, int end, float seconds )
    {
        if (playing) return;

        stopped = false;
        if (looping) return;

        looping = true;
        this.start = start;
        this.end = end;

        currentCell = start;
        float difference = (float)end - (float)start;

        timeShift = seconds / difference;
    }

    public void Stop()
    {
```

```
    if (playing) return;

    stopped = true;
    looping = false;
    totalTime = 0.0f;
    timeShift = 0.0f;
}

public void GotoFrame(int number)
{
    if (playing) return;

    if (number < 0 || number >= cellList.Count) return;
    currentCell = number;
}

public void PlayAll(float seconds)
{
    if (playing) return;
    GotoFrame(0);
    stopped = false;
    looping = false;
    playing = true;
    start = 0;
    end = cellList.Count - 1;

    timeShift = seconds / (float)cellList.Count;
}

public void Play(int start, int end, float seconds)
{
    if (playing) return;
    GotoFrame(start);
    stopped = false;
    looping = false;
    playing = true;
    this.start = start;
    this.end = end;

    float difference = (float)end - (float)start;

    timeShift = seconds / difference;
}


public void Draw(SpriteBatch batch)
```

```
    {
        if (cellList.Count == 0 || currentCell < 0 ||
            cellList.Count <= currentCell) return;

        batch.Draw(cellList[currentCell].cell, position, null, Color.White, 0.0f,
            spriteOrigin,
            new Vector2( scale, scale), spriteEffect, 0.0f );
    }

    public void Update(GameTime gameTime)
    {
        if (stopped) return;

        totalTime += (float) gameTime.ElapsedGameTime.TotalSeconds;
        if (totalTime > timeShift)
        {
            totalTime = 0.0f;

            currentCell++;

            if (looping)
            {
                if (currentCell > end) currentCell = start;
            }
            if (currentCell > end)
            {
                currentCell = end;
                playing = false;
            }
        }
    }
}
```

Like with the background code we won't go into much of the inner details, but will hit the main points. The constructor for the Animation class takes in a position, which is where on the screen the animation should be drawn:

```
public Animation(Vector2 position)
```

You can also set the spriteOrigin and change the position later, for example if playerAnimation is our object:

```
playerAnimation.SpriteOrigin = new Vector2(50.0f, 50.0f);
playerAnimation.Position.X = 50.0f;
```

are both valid lines of code. To create the animation cells we just load a texture and add it to it:

```
Texture2D animationCell = Content.Load<Texture2D>("Walk_1");
playerAnimation.AddCell(animationCell);
```

Right now the Animation class doesn't keep an id or numbering of the cell, so when referring to the cells later it is by the number of the order it was added to the animation. The first cell you add is zero (it is zero based indexed) the second cell is one, and so on. To draw a specific cell we use:

```
public void GotoFrame(int number)
```

where number is the frame number to draw. To play an animation we have two options, to play an animation once or to play the animation and have it loop. The syntax for the two are similar:

```
public void Loop(int start, int end, float seconds )
public void Play(int start, int end, float seconds)
```

where start is the number of the first frame of the animation to play and end is the last. The seconds is how much time it should take to play the animation. If you look at the Game1.cs you'll find that the Loop is used for the walk cycle, since we want to repeat it as long as the character is walking. But for the attack animation Play is used because we want to play the animation just once during an attack. In addition to these there are LoopAll and PlayAll methods that cycle through every cell for the animation.

Two more methods let you flip which way the pictures are facing:

```
public void SetMoveLeft()
public void SetMoveRight()
```

If you look through the Game1.cs in the Animation sample you'll see an example of how the animation works. Pressing the left or right arrow will change the player position onscreen to the left or right and play the walk animation, but if neither are pressed the animation goes to frame zero, the standing still picture. Looking at this sample should be enough to create standard scroller games with the character walking around the screen.

Adding Animation to Ship Fighter

Now let's add some animation to the *Ship Fighter* game. What we'll do is pretty simple; we'll add in two more pictures, one of the ship rotating to the left and another rotating to the right, so when we turn left or right we'll see the ship change. The solution for this project is Player with Animation at xnagamemaking.com. Or you can follow these steps to add to the background project from last time.

The first thing to do is to add Animation.cs to the ship project. This is done the same as for MultiBackground.cs, copy Animation.cs to the folder with the Game1.cs and from the solution explorer right click on the project name and select Add->Existing Item and select the Animation file. Also to the content directory copy the art files shipLeft and shipRight and add them to the project.

The first thing we'll do is in the Player.cs delete the single sprite that used to hold our ship picture and replace it with an Animation object. So delete the line in Player.cs that has:

```
Texture2D shipSprite;
```

and replace it with an Animation object:

```
Animation playerAnimation;
```

You'll also need to add the using xnaExtras to the Player.cs. Currently the constructor for the Player class takes in a texture and saves it and calculates the sprite origin from it:

```
public Player(GraphicsDevice Device, Vector2 position, Texture2D sprite)
{
    this.position = position;

    shipSprite = sprite;
    spriteOrigin.X = (float) shipSprite.Width / 2.0f;
    spriteOrigin.Y = (float)shipSprite.Height / 2.0f;
    windowHeight = Device.Viewport.Height;
    windowWidth = Device.Viewport.Width;
}
```

Since we are no longer using a single sprite we won't pass a Texture2D in but we will need a sprite origin. So we'll change it to:

```
public Player(GraphicsDevice Device, Vector2 position, Vector2 origin )
{
    // The position that is passed in is now set to the position above
    this.position = position;

    // Create the animation, this method is in Animation.cs
    playerAnimation = new Animation(position);

    windowHeight = Device.Viewport.Height;
    windowWidth = Device.Viewport.Width;

    playerAnimation.SpriteOrigin = origin;
    playerAnimation.Stop();
}
```

Here we initialized the animation by giving it the start position for the player and set the SpriteOrigin for the animation. The next thing we'll do is add a method to add frames of animation to the player. So after the constructor add the following to the player class.

```
public void AddCell(Texture2D cellPicture)
{
        playerAnimation.AddCell(cellPicture);
}
```

Now we're setup to create a Player and give it the different frames of animation for the ship. We'll do that now. In the Game1.cs LoadContent method remove the current player initialization and replace it with the following:

```
Texture2D shipMain = Content.Load<Texture2D>("shipSprite");
// Create our player, besides position we pass it the graphics device
playerShip = new Player(graphics.GraphicsDevice, new Vector2(400.0f,
    350.0f), new Vector2(shipMain.Width / 2, shipMain.Height / 2));

playerShip.AddCell(shipMain);
Texture2D shipLeft = Content.Load<Texture2D>("shipLeft");
playerShip.AddCell(shipLeft);
Texture2D shipRight = Content.Load<Texture2D>("shipRight");
playerShip.AddCell(shipRight);
```

This modification first reads in the original ship sprite, then creates a new player (getting the sprite origin from the ship sprite.) After that two more frames of animation are read in; this means our ship animation has three frames. The original ship at frame zero, left is frame one and right is frame two.

Now we have the player animation defined, we need to modify the player update and draw methods to handle the animation. Let's first modify the draw method. Erase the batch drawing routine and replace it with this:

```
public void Draw(SpriteBatch batch)
{
    playerAnimation.SetPosition(position);
    playerAnimation.Draw(batch);
}
```

Here we just tell the animation object to draw. But we want to make sure the animation is drawn at the correct position, so before we draw it we just update the position. This is all we need to do for the drawing

modification, next update the Update loop. The first thing to do is just to add the playerAnimation update to the update loop. In the player Update method add the following line:

```
playerAnimation.Update(gameTime);
```

This is the only thing we need to do to the update loop, but we need to modify the move left and right methods to switch the animation frame. So we'll change the TurnRight() and TurnLeft() to:

```
public void TurnRight()
{
    playerAnimation.GotoFrame(2);
    position.X += 3.0f;
}

public void TurnLeft()
{
    playerAnimation.GotoFrame(1);
    position.X -= 3.0f;
}
```

This flips the frames when turning, but now we need a way to flip back to the original ship sprite. So we'll add the following method that resets that we're going straight:

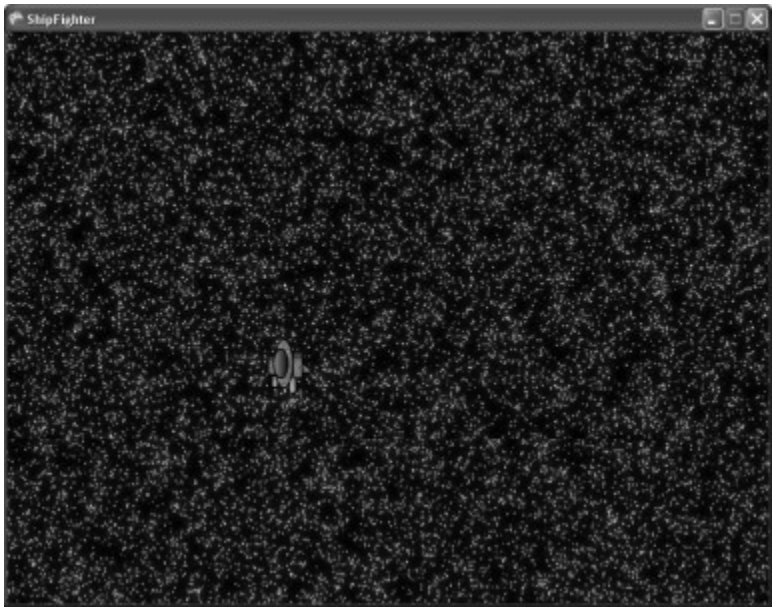
```
public void GoStraight()
{
    playerAnimation.GotoFrame(0);
}
```

Now let's modify the Game1.cs to reflect this new method. All we'll do is when testing if the left and right buttons are pressed if they're not then we'll tell the player to go straight:

```
if (keyState.IsKeyDown(Keys.Left)
        || gamePadState.DPad.Left == ButtonState.Pressed)
{
```

```
    playerShip.TurnLeft();
}
else if (keyState.IsKeyDown(Keys.Right)
            || gamePadState.DPad.Right == ButtonState.Pressed)
{
    playerShip.TurnRight();
}
else
    playerShip.GoStraight();
```

Now if you run the program you'll see the ship change sprites when moving left and right.



Summary

We now have some basic animation in our ship fighter game and have seen how to create a character with animation. In the next chapter we'll make things more complex by giving the player the ability to fire and adding enemies to the game.

Chapter 10: Adding Firepower and Creating Enemies

We have all of the basic elements to create a 2D game now: We know how to draw sprites, even animate them, and have them move around based on player input. We've also looked at how to make scrolling backgrounds to make our game more complete. But so far we've kept everything simple by having one basic entity, the player. In this chapter we'll start putting more entities into the game, first putting in fire balls the player can fire and then some enemy ships for the player to interact with. We will cover:

- How to put in simple projectiles the player can shoot
- How to put in enemies that interact with the player.
- What collision detection is and how it works in 2D.

The complete project for this chapter is *ShipFighter with Enemies* at xnagamemaking.com, but in our descriptions we'll be taking from the ShipFighter Animation project from last chapter.

To Shoot Fireballs

The first thing we'll do is give the player the ability to shoot fireballs. Whenever the player presses the firing key (spacebar or trigger) we'll create a fireball at the ship's location and have it travel up the screen. We'll have a separate class for the fireball objects, it will be simple and just draw them and move them up. We'll also keep a list of all the fireball objects in the game, and when the player fires it will create a new fireball and add it to the list. Here is the listing for our Fireball class:

```
class Fireball
{
    Vector2 position;

    Texture2D picture;
```

```
    float speed;

    public Fireball(Texture2D firePicture, Vector2 startPosition, float
      updateSpeed)
    {
       picture = firePicture;
       position = startPosition;
       speed = updateSpeed;
    }

    public Vector2 Position{ get{ return position; } }

    public void Draw(SpriteBatch batch)
    {
       batch.Draw(picture, position, null, Color.White, 0.0f, new Vector2(
          10.0f, 10.0f ), 1.0f, SpriteEffects.None, 1.0f);
    }

    public void Update(GameTime gameTime)
    {
       position.Y -= speed *
          (float)gameTime.ElapsedGameTime.TotalSeconds;
    }
}
```

When we create a fireball we just pass in a picture of it to draw, a start position, and a speed. At each update we just move the fireball up the screen by it's current speed (moving up is subtracting from the Y in the position). And the drawing method just draws the fireball picture at it's current position.

Next we'll add a list to hold all of the fireballs that are running in a game, so in the Game1.cs add the following variable:

```
List<Fireball> playerFireballList = new List<Fireball>();
```

Remember our four steps for adding something to the Game object: declare it, initialize it, draw it and update it. The above line declares and initializes the list, next we'll add it to the Update loop. In the Update method in the Game add the following:

```
for (int i = 0; i < playerFireballList.Count; i++)
{
    playerFireballList[i].Update(gameTime);
    if (playerFireballList[i].Position.Y < -100.0f)
        playerFireballList.RemoveAt(i);
}
```

This addition just calls the update for every fireball and checks to see if it went off screen. If the fireball is above the screen and no longer visible we delete it. Then in the Draw loop of the game object add this:

```
foreach (Fireball f in playerFireballList)
    f.Draw(spaceBatch);
```

Which just goes through every fireball in the list and draws it. We've setup a list to add fireballs too but we're not adding them in yet. What we'll do is whenever we press a fire button on the keyboard or gamepad we'll create a fireball at the current player position. But to create fireball's we'll need a texture for it, so add the file playerFireBall.png to the project. Then put a variable for the sprite in the Game object:

```
Texture2D playerFireballTexture;
```

and initialize it in the LoadContent method:

```
playerFireballTexture = Content.Load<Texture2D>("playerFireball");
```

Since we need the player position for the fireball creation we'll need to add a property to the Player class so we can access the position from our game class. In the Player.cs add the following below the position definition:

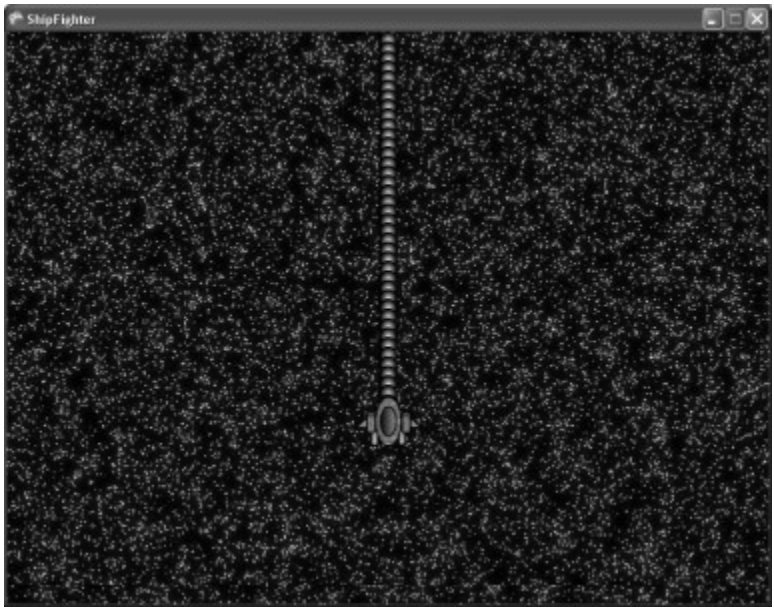
```
public Vector2 Position
{
        get
```

```
        {
            return position;
        }
}
```

Now we can get the current player position. We'll add another key press test to the UpdateInput in the Game class that will create a fireball and add it to the list:

```
if (gamePadState.Buttons.A == ButtonState.Pressed ||
            gamePadState.Triggers.Right >= 0.5f ||
            keyState.IsKeyDown(Keys.Space))
{

    Fireball shot = new Fireball(playerFireballTexture,
        new Vector2(playerShip.Position.X, playerShip.Position.Y - 10.0f),
        300.0f);

     playerFireballList.Add(shot);
}
```

If you run the program now and try firing with the spacebar you'll see something like the following:



What we're seeing here goes back to the keyboard events not being event driven, but polling. When we press the fire button to create a fireball we might hold it down for maybe half a second. But our game loop is updating at sixty times per second, so holding the key down for half a second will create about thirty fireballs. This is because we are just polling the keyboard and gamepad at each update. One common solution to this is to save the previous Keyboard state, so when we update we can see if the key was pressed in the last loop update or if it is a new key press. But we'd like to have something here where if the player holds down the fire button the player will continue to attack, like firing every three seconds. To do this we'll create a buttonFireDelay variable. It will work by at every update we'll subtract from it how much time has passed since the last update and when we press the attack we'll set it to three seconds. We also won't let the keyboard fire key be registered until three seconds is up. This means when the fire button is pressed the buttonFireDelay will be set to three seconds, and we won't be able to fire again until three seconds is up. So in the Game class add the following variable:

```
float buttonFireDelay = 0.0f;
```

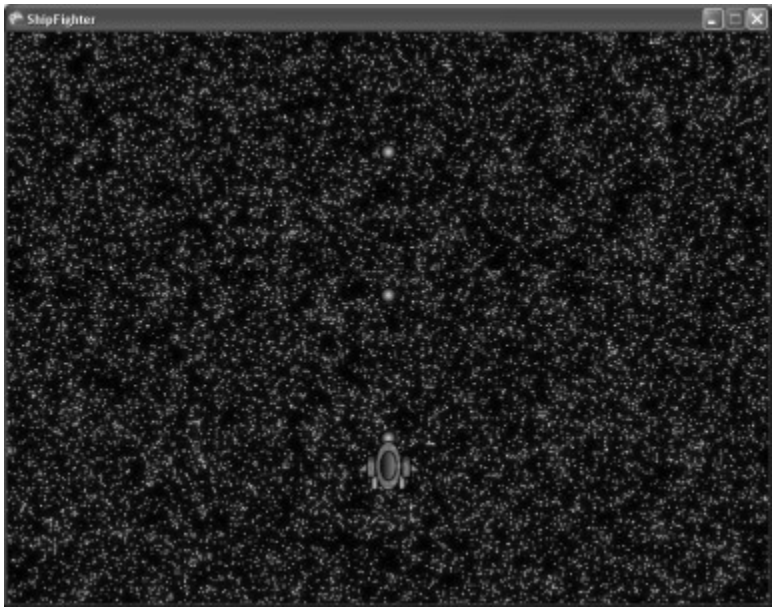
And in the update loop we'll get how much time has passed since the last update and subtract it from the buttonFireDelay. Add the following to the Update loop:

```
float elapsed = (float)gameTime.ElapsedGameTime.TotalSeconds;
buttonFireDelay -= elapsed;
```

Then change the UpdateInput attack part to:

```
 if (gamePadState.Buttons.A == ButtonState.Pressed ||
            gamePadState.Triggers.Right >= 0.5f ||
            keyState.IsKeyDown(Keys.Space))
{
    if (buttonFireDelay <= 0.0f)
    {
        Fireball shot = new Fireball(playerFireballTexture,
            new Vector2(playerShip.Position.X, playerShip.Position.Y - 10.0f),
            300.0f);

        playerFireballList.Add(shot);
        buttonFireDelay = 0.25f;
    }
}
```

In this modification we only allow the player to fire if the buttonfireDelay is less than or equal to zero. When the player does fire the buttonFireDelay is set to 0.25 so we won't be able to fire again for a quarter of a second. If you run the game now the player will fire normally:



That sets up our player firing. Next we'll give the player something to shoot at.

Creating Enemies

One of the largest issues when designing games are the enemy characters the player interacts with. Usually AI is a big issue for enemies as is collision detection, which we'll talk more about later in the chapter. For the moment we'll just make some basic enemies to interact with.

Our enemies will be pretty simple, just sprites that comes down the screen. They'll also have the ability to move left and right as they travel down, and fire at the player. For the collision detection we'll test if the player's fireballs hit an enemy or if an enemy or enemy's fires hit the player. (Note that two extra features originally added for the enemies were an inheritance hierarchy for the enemies: having a base enemy class that other more specific enemy types inherited from (this is an object-oriented way of doing things.) The other feature was reading in an xml file that had info for all of the enemies, when and where they are created for the game. But putting in the class inheritance hierarchy and reading xml was difficult for several students and slowed things down, so

versions of the ShipFighter game with these features would be a good exercise but we won't discuss them in the text.)

Enemy Project

Now we'll start adding the enemies to our project. The first thing to do is add some new resources for the enemies. Add the pictures blueEnemy.png, greenEnemy.png, and enemyFireball.png to the Content folder of the ShipFighter project. Then create a new source file for the enemies, we'll call Enemy.cs. Here is the code for the enemy class:

```
class Enemy
{
    Vector2 position;

    Texture2D picture;

    float speed = 150.0f;

    float deltaX = 0.0f;
    float xLength = 0.0f;
    float xStart = 0.0f;

    bool firingActive = false;
    bool firing = false;
    float fireSpeed = 1.0f;
    float totalTime = 0.0f;

    public bool FiringActive
    {
       set { firingActive = value; }
    }

    public bool Firing
    {
       set { firing = value; }
       get { return firing; }
    }

    public Enemy(Texture2D picture, Vector2 startPosition, float speed )
    {
       this.picture = picture;
```

```
            position = startPosition;

            this.speed = speed;
        }

        public void SetAcrossMovement( float deltaX, float xLength )
        {
            this.deltaX = deltaX;
            this.xLength = xLength;
            xStart = position.X;
        }

        public Vector2 Position {
            get { return position; }
        }

        public void Draw(SpriteBatch batch)
        {
            batch.Draw(picture, position, null, Color.White, 0.0f, new Vector2(
                40.0f, 20.0f ), 1.0f, SpriteEffects.None, 0.0f);
        }

        public void Update(GameTime gameTime)
        {
            float elapsed = (float)gameTime.ElapsedGameTime.TotalSeconds;
            position.X += deltaX * elapsed;

            if (Position.X < xStart - xLength || Position.X > xStart + xLength)
                deltaX *= -1.0f;

            position.Y += speed * elapsed;

            if (firingActive)
            {
                totalTime += elapsed;

                if (totalTime > fireSpeed)
                {
                    totalTime = 0.0f;
                    firing = true;
                }
            }
        }
}
```

Let's go through this class. The first variables we have for the enemy are pretty straightforward:

```
Vector2 position;

Texture2D picture;

float speed = 150.0f;
```

Just a position on the screen where the enemy currently is, a Texture2D to hold the sprite for the enemy, and a speed (still in pixels per second) of how fast the enemy is moving down the screen. These variables are set in the constructor for the enemy:

```
public Enemy(Texture2D picture, Vector2 startPosition, float speed )
{
    this.picture = picture;

    position = startPosition;

    this.speed = speed;
}
```

When we create an enemy we have to give it a sprite, start position, and speed. The next few variables are for moving the enemy left and right across the screen:

```
float deltaX = 0.0f;
float xLength = 0.0f;
float xStart = 0.0f;
```

The deltaX is how much the enemy moves horizontally at each update (again in pixels per second.) The next two variables handle how far across the screen it can move. We don't want the enemy to move completely left and right across the entire screen, so we have xLength, which is how far it can move left or right and xStart which is the start position for the x. So if an enemy has an xStart of 200 and an xLength of 50 it will move left and right across the screen between 150 and 250 (at a speed of

deltaX). You can see how this works in the Update method for the enemy, which we'll cover in a moment. The variables are set in the SetAcrossMovement method:

```
public void SetAcrossMovement( float deltaX, float xLength )
{
    this.deltaX = deltaX;
    this.xLength = xLength;
    xStart = position.X;
}
```

Note the xStart is just the current position of the enemy when this method is called.

The last few variables are to handle the firing of fireballs (or in this case lasers) by the enemy:

```
bool firingActive = false;
bool firing = false;
float fireSpeed = 1.0f;
float totalTime = 0.0f;
```

First we have two bools about firing. The firingActive is if the current enemy can fire at all. If it is true the enemy will start firing and if it's false it won't fire at all. The bool "firing" is if the enemy has firingActive, if it is firing at the current update. If an enemy has firingActive then firing will toggle true and false based on if the enemy is currently firing. The fire speed is what frequency the enemy is firing at. At 1.0f this means the enemy will be firing once a second. The totalTime is just an internal counter for timing the firing. We activate firing for an enemy through the following property:

```
public bool FiringActive
{
    set { firingActive = value; }
}
```

We just set this true or false from somewhere in the game to let an enemy start firing or not. The next property has to do when we fire:

```
public bool Firing
{
    set { firing = value; }
    get { return firing; }
}
```

Note there's no method here to actually fire. When we have an enemy fire it will create a new fireball object inside of our Game, but not in the enemy object. So in the Game object we'll test at each update if the enemy if firing. If it is we'll create a new fireball there and set the firing flag here back to false. This will make more sense when we look at the Game1.cs in a bit.

That sets up the basic variables for our enemies, the next things we need are a draw method and an update one. For the drawing we'll just take in a spriteBatch and use it to draw the enemy at its current position:

```
public void Draw(SpriteBatch batch)
{
    batch.Draw(picture, position, null, Color.White, 0.0f, new Vector2( 40.0f,
        20.0f ), 1.0f, SpriteEffects.None, 0.0f);
}
```

The Update loop will be a little more complex. The first line just gets the amount of time that has passed since the last update in seconds, to use in our movement calculations:

```
float elapsed = (float)gameTime.ElapsedGameTime.TotalSeconds;
```

Next we'll move the player left or right using the deltaX and the elapsed time:

```
position.X += deltaX * elapsed;
```

Then we'll test if the player has moved enough to the left or right to warrant switching directions:

```
if (Position.X < xStart - xLength || Position.X > xStart + xLength)
            deltaX *= -1.0f;
```

Multiplying deltaX by negative one changes its sign and causes it to move the opposite direction.

Next move the enemy down the screen based on its speed:

```
 position.Y += speed * elapsed;
```

That takes care of the movement of the enemy. Next we need to calculate if it should be firing or not:

```
if (firingActive)
{
    totalTime += elapsed;

    if (totalTime > fireSpeed)
    {
        totalTime = 0.0f;
        firing = true;
    }
}
```

This is really simple. We just say if the enemy is currently firing (firingActive is true) then add the time that has elapsed from this update to the current time. If this total time is more than the fireSpeed than we should fire, which just means setting firing to true. Then we reset the totalTime to zero so we can start counting again.

This is the class for our enemies, now let's add them to the Game. We're going to be adding two basic enemy types to the game, a green enemy

type that moves across the screen and a blue enemy type that fires at the player. In the Game1.cs before the player class go ahead and put the following enum in:

```
public enum EnemyType
{
    green,
    blue
}
```

This enum will be handy when creating enemies since it will make clear if we are talking about the green or blue kind. Next let's add some enemies to Game1.cs:

Remember the four steps to adding something to our game:

1. Declare it
2. Initialize it
3. Add it to the Draw loop
4. Add it to the Update Loop

So first we'll declare some variables for our enemies. We'll need a list to hold all of our enemies and a list to hold all of the enemy fireballs that the blue ones shoot. We'll also need textures to hold the sprites for the green enemy, blue enemy, and the enemy fireballs (and we'll throw in a random number generator called rnd for use later). In the game class add the following variables:

```
List<Fireball> enemyFireballList = new List<Fireball>();

// List of enemies
List<Enemy> enemyShipList = new List<Enemy>();
Texture2D enemyTextureGreen;
Texture2D enemyTextureBlue;
Texture2D enemyFireballTexture;
Random rnd = new Random();
```

The two lists are initialized there, but we need to load in the textures. To initialize the sprites in the LoadContent method add the following:

```
enemyTextureGreen = Content.Load<Texture2D>("greenEnemy");
enemyTextureBlue = Content.Load<Texture2D>("blueEnemy");
enemyFireballTexture = Content.Load<Texture2D>("enemyFireball");
```

We have everything initialized, so let's draw them. To draw the enemies we'll go through each item in the enemy list and enemy fireball list and call their draw method:

```
 foreach (Fireball f in enemyFireballList)
    f.Draw(spriteBatch);

foreach (Enemy e in enemyShipList)
    e.Draw(spriteBatch);
```

That's pretty straightforward, next we need to add them to the Update loop. To update the enemies and enemy fireballs we'll just call their update method, and like the player fireballs we'll check both of them to see if they've gone off screen and out of the game. If they're out of the game we'll delete them from their list. And we'll also want to create any enemy fireballs if any of the enemies are firing. So for each enemy if it is firing we'll create a new enemy fireball and add it to the list:

```
for (int i = 0; i < enemyShipList.Count; i++)
{
    enemyShipList[i].Update(gameTime);

    if (enemyShipList[i].Firing)
    {
        Fireball fireball = new Fireball(enemyFireballTexture,
                new Vector2(enemyShipList[i].Position.X + 10.0f,
        enemyShipList[i].Position.Y + 30.0f), -300.0f);
        enemyFireballList.Add(fireball);
        enemyShipList[i].Firing = false;
    }
```

```
    if (enemyShipList[i].Position.Y > 900.0f)
                enemyShipList.RemoveAt(i);
}

for (int i = 0; i < enemyFireballList.Count; i++)
{
    enemyFireballList[i].Update(gameTime);

    if (enemyFireballList[i].Position.Y > 900.0f)
        enemyFireballList.RemoveAt(i);
}
```

There we have it, all the enemies are setup for use in the game. Now we need a way to create them. For the moment we'll just use keys on the keyboard to create them. If the C key is pressed we'll create a green enemy and if the D key is pressed we'll create a blue. We'll need a function that creates enemies, so add the following to the Game class:

```
public void CreateEnemy(EnemyType enemyType)
{
    Random r = new Random();
    int startX = r.Next(800);
    if (enemyType == EnemyType.green)
    {
        Enemy enemy = new Enemy(enemyTextureGreen, new
            Vector2(startX, -100), 150.0f);
        enemy.SetAcrossMovement((float)startX / 800.0f * 250.0f, 50.0f);
        enemyShipList.Add(enemy);
    }
    if (enemyType == EnemyType.blue)
    {
        Enemy enemy = new Enemy(enemyTextureBlue, new Vector2(startX,
            -100), 150.0f);
        enemy.Firing = true;
        enemyShipList.Add(enemy);
    }
}
```

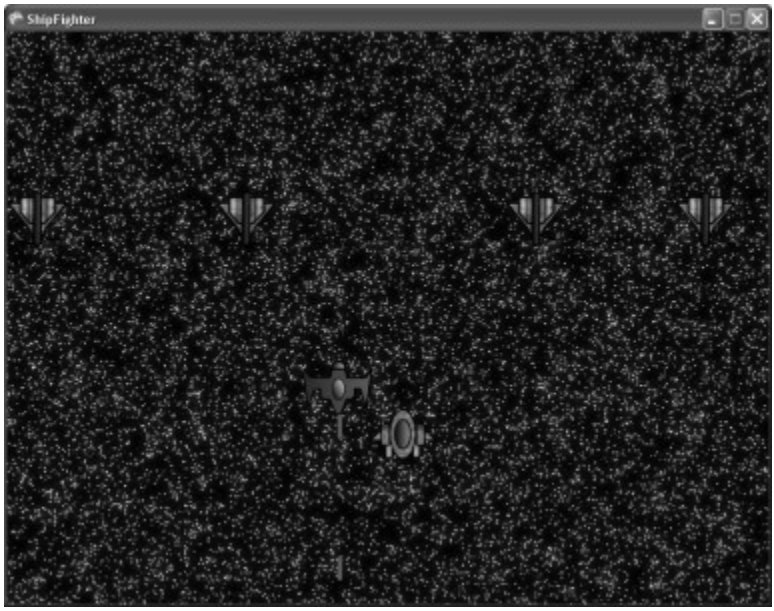
This function takes in an EnemyType type (green or blue from the enum.) Then it creates a random number for the x start position:

```
int startX = rnd.Next(800);
```

The rnd is a Random number generator and the .Next(800) means to return a number in the range 0-800. Then we use this start position for the X and create a new enemy. We'll create either a green enemy that we set to move left and right across the screen using the SetAcrossMovement we described earlier or a blue enemy that we set FiringActive to true to get it to start firing. Now we just need to add some more cases to the updateInput function:

```
if (keyState.IsKeyDown(Keys.C))
    if (buttonFireDelay <= 0.0f)
    {
        CreateEnemy(EnemyType.green);
        buttonFireDelay = 0.25f;
    }

 if (keyState.IsKeyDown(Keys.D))
    if (buttonFireDelay <= 0.0f)
    {
        CreateEnemy(EnemyType.blue);
        buttonFireDelay = 0.25f;
    }
```

These two just check if the C or D is pressed and creates an enemy. If you run the game and hit the C or D keys you'll see enemies fill up the screen.



This all looks nice enough, but notice if you move the player ship around it isn't colliding with anything, everything is acting very independently. We need to put in collision detection, which we'll do next.

Collision Detection

Collision detection is a very famous problem, not just for video games, but for computer science in general. The problem is given a collection of moving objects, test if any of them have collided (or will collide) with each other. This sounds simple enough, but in practice it can become very difficult; in a typical game a huge percent of the CPU time is spent on collision detection. Entire books have been written on the subject. We'll just talk about a few basic concepts here.

The first thing to know about collision detection is when we test to see if two objects intersect we don't do a full detail test first. If we have two 3D models, each with thousands of polygons doing a full test (testing each polygon to see if intersects with any other polygon) to see if the models intersect would be very expensive. We first surround each object with a

simple geometric shape, and then test the basic shape to see if they collide. One of the popular simple shapes is a bounding sphere. If each object is contained in a sphere then collision detection becomes simpler. To calculate an intersection we just test the distance between the two objects and see if it is less than the sum of their radii:

```
if( (position1 – position2).Length() > radius1 + radius2 )
    true if they intersect
```

Another popular method is to use boxes, or more specifically axis-aligned bounding boxes (aabbs). An aabb is a large box that contains an object, with each edge of the box parallel with an axis of the coordinate system. This kind of collision detection is good too, but we'll use the bounding spheres.

Getting back to our game, we're going to do three things with the collision detection:

Detect if the enemy is hit by a player fireball, and if so delete the enemy and the fireball.

Detect if the player is hit by a an enemy fireball, if it is have it blink for a few seconds.

Detect if the player collides with an enemy. If so delete the enemy and have the player blink for a few seconds.

The player blinking is the player going into a recovery state. Later when we track the number of lives the player has we'll set it up so if the player gets hit it loses a life but then becomes invincible for a few seconds, which we'll show by having it blink.

First let's modify the enemy class so it will take in a list of player fireballs and see if any of them hit it. To do this we'll need a radius of the enemy ship, so add the following variable to the enemy class:

```
float radius = 40.0f;
```

```
public float Radius
 {
    get { return radius; }
}
```

Why do we set a radius of forty ourselves, instead of calculating it from the sprite? Sometimes we want to adjust the collision detection to make it a bit more sensitive so things collide easier or looser to make the collisions more strict (such as wanting the fireball to have to hit an enemy near the center of it to count as a collision.) For the fireballs we'll treat them as points, we won't bother setting any radius for them. Add the following to the enemy class:

```
public int CollisionBall( List<Fireball> fireballList )
{
    for( int i = 0; i < fireballList.Count; i++ )
    {
        if ((fireballList[i].Position - position).Length() < radius)
            return i;
    }

    return -1;
}
```

This takes in fireball list and then checks each one to see if it collided with the player. If a fireball does intersect with the player it returns the index into the list of the fireball, otherwise it returns -1. In the update method in the Game class where we iterate through every enemy we use this by adding the following:

```
int collide = enemyShipList[i].CollisionBall(playerFireballList);

if (collide != -1)
{
    enemyShipList.RemoveAt(i);
    playerFireballList.RemoveAt(collide);
}
```

This does what we said, it gives the enemy the playerFireballList and if one of them hit it will delete the fireball and the enemy. This takes care of the first case, next we need to handle the player colliding with an enemy ship. To do this we need to add a function to test collision to the player class, and the functionality to make it go into a recovering state. First we'll add:

```
float radius = 50.0f;

float blinkTime = 0.0f;
float blinkTimeTotal = 0.0f;
bool blinkOn = false;
bool recoveringActive = false;
const float recoverLength = 3.0f;
```

The radius is used for collision detection, the rest for the recovering state. The recovering state will work like this: when the player collides with something the recoveringActive bool will be set to true and it will stay true for the amount of time in recoverLength. While we are recovering the blinkOn bool will represent if we are currently in the on or off blink state and will decide if we should draw the player. The blinkTime will track how much time has passed since we last changed blinkOn, when it gets to a quarter second we toggle the blinking. The blinkTotalTime tracks how much time has passed when we started recovering and when it hits the recoverLength we'll set the recovering to false. This is clearer if you look at the new code for the Update method for the player:

```
if (recoveringActive)
{
    const float blinkTimeSlice = 1.0f / 15.0f;
    float elapsed = (float)gameTime.ElapsedGameTime.TotalSeconds;
    blinkTime += elapsed;
    blinkTimeTotal += elapsed;
    if (blinkTimeTotal > recoverLength)
        recoveringActive = false;
    else
    {
        if (blinkTime > blinkTimeSlice)
        {
```

```
            blinkOn = !blinkOn;
            blinkTime = 0.0f;
        }
    }
}
```

Here we take the time from the last update and add it to the blinkTime and blinkTimeTotal. If blinkTimeTotal is greater than the recovery length we turn the recovering (blinking) off. Else we just test if enough time has passed to toggle the current state of the blink – visible or invisible.

Next we need a method for the player to test for collisions with other objects. To the Player class we add:

```
public bool CollisionTest(Vector2 point, float radius)
{
    if ((point - position).Length() < this.radius + radius)
    {
        if (!recoveringActive)
        {
            recoveringActive = true;
            blinkTimeTotal = 0.0f;
        }
        return true;
    }
    return false;
}
```

This method takes in a point and a radius and does our basic bounding sphere calculation to test for a collision. If we have a collision and we are not currently in a recovery state we send the player into the recovering state by setting recoveringActive to true and resetting the blinkTimeTotal back to zero. The function returns if we had a collision true or false.

To use this for if the player gets hit with an enemy fireball we'll change the following to the main Update loop in our Game.cs:

```
for (int i = 0; i < enemyFireballList.Count; i++)
{
    enemyFireballList[i].Update(gameTime);
    playerShip.CollisionTest(enemyFireballList[i].Position, 20.0f);
}
```

In this update we go through every enemy fireball and test to see if it collided with the player. The player's collision test will send it into its recovery state if it has.

Lastly we'll test to see if any enemy collided with the player, and if so delete it and send the player into the recovery state. To do this we'll add a little to our enemy collision code from the Update method in the Game:

```
int collide = enemyShipList[i].CollisionBall(playerFireballList);

if (collide != -1)
{
    enemyShipList.RemoveAt(i);
    playerFireballList.RemoveAt(collide);
}
else
if (playerShip.CollisionTest(enemyShipList[i].Position,
enemyShipList[i].Radius))
{
    enemyShipList.RemoveAt(i);
}
else if (enemyShipList[i].Position.Y > 2000.0f)
    enemyShipList.RemoveAt(i);
```

This is the same as before, but with the addition of the else clause. Here we test each enemy to see if it hits the player, if it does it gets deleted and the player's CollisionTest takes it to the recovery state. We also put in a little test to see if the enemy is far off screen, and if it is we won't see it in the game anymore and we'll delete it.

If you run the game you'll find all of the collision detection working. When we fire at enemies they disappear and the player reacts to being hit

by other enemies. One last thing to do is to have the enemies be created in a slightly better way than just through keyboard presses.

One simple way to do this is to just use a timer function. We'll just track how much time has passed in the game total and at certain intervals create new enemies. Specifically, in the game class we'll add the following variable:

```
float totalTime = 0.0f;
```

and in the Update loop we'll add to it:

```
 float elapsed = (float)gameTime.ElapsedGameTime.TotalSeconds;
          totalTime += elapsed;
```

Then we'll add a little function:

```
public void CreateEnemyTime()
{
    if (totalTime > 1.0f && totalTime < 1.01f)
        CreateEnemy(EnemyType.blue);

    if (totalTime > 2.2f && totalTime < 2.21f)
        for (int i = 0; i < 4; i++)
            CreateEnemy(EnemyType.green);

    if (totalTime > 7.2f && totalTime < 7.21f)
        for (int i = 0; i < 7; i++)
            CreateEnemy(EnemyType.blue);
}
```

And add a line in the Update loop to call it. This is pretty simple, just checking the total time and creating enemies when enough time has passed. Again, look on xnagamemaking.com to see a more advanced version of this.

Summary

That wraps up putting enemies in the game. We almost have a fully featured game, but there are still a few things to take care of. In the next chapter we'll add some splash screens, HUD text, and explosion effects.

Chapter 11: Adding HUD Text, Splash Screens, and Special Effects

Our game is getting close to being complete. But there are still a few elements that are important and need to go in the game. The first is adding some text to our display. We'll add a little point counter and live counter. We'll also need a simple menu system, which for now will be just a few splash screens. Lastly we want to add in some special effects, and we'll add an explosion effect using particle systems. The goals for this chapter are:

- Learn how to draw text on a HUD (Heads Up Display.)
- Add some simple splash screens to the game.
- Learn what particle systems are and use them to create explosion effects.

Adding HUD Text

What we mean by adding HUD text is pretty straightforward, we are just going to go over how to add some text displayed at various parts of the screen. The way we'll do this is to create an object called a Sprite Font. This object reads in a font from our system and uses it to create text in our game (which means you have to be careful about legal issues, as fonts on your system are copyrighted and may not be distributable. Check at Microsoft's XNA's sight for more details.)

To add text to the screen we'll do the following (which is adapted from the XNA documentation):

Right-click on the Content folder in Solution Explorer, click Add, and then click New Item. Choose Sprite Font from the Add New Item dialog box. Name the Sprite Font ShipFont, since it will be the only font we're using for the game. XNA creates the font and double click it to open it. You'll see an xml file with several fields:

```
<!--
  Modify this string to change the font that will be imported.
  -->
  <FontName>SpriteFont1</FontName>

  <!--
  Size is a float value, measured in points. Modify this value to change
  the size of the font.
  -->
  <Size>14</Size>

  <!--
  Spacing is a float value, measured in pixels. Modify this value to change
  the amount of spacing in between characters.
  -->
  <Spacing>2</Spacing>

  <!--
  Style controls the style of the font. Valid entries are "Regular", "Bold",
"Italic",
  and "Bold, Italic", and are case sensitive.
  -->
  <Style>Regular</Style>
```

This is the file XNA parses to create text for the screen. In the FontName element where it says SpriteFont1 delete that and put in Courier New. This is the font we'll be using for the ship game, but you can put the name of any font you have on your machine. Then they are a few other elements to describe the text you can change, such as the size, spacing, and if it should be bold or italic. We'll leave these to their defaults, but play around with them to see different text effects.

Next let's add the sprite to our game. But before that we'll add two more variables, one in the player class and one in the game class for the information we want to display. For the player we'll keep track of lives. We'll just create a variable called lives, set it to a default number (5) and every time the player is hit by something we'll subtract one from it. So int the Player.cs add the following:

```
 // Total number of lives
int lives = 5;

public int Lives
{
    set { lives = value; }
    get { return lives; }
}
```

We put in the Lives property so we can access the number of lives from outside the player class. Next we'll change the CollisionTest method to the following:

```
public bool CollisionTest(Vector2 point, float radius)
{
    if ((point - position).Length() < this.radius + radius)
    {
        if (!recoveringActive)
        {
            lives--;
            recoveringActive = true;
            blinkTimeTotal = 0.0f;
        }
        return true;
    }
    return false;
}
```

The only difference here from the original is the addition of the line lives--; So when we are hit with something the number of lives now gets decremented once.

Next we'll keep track of the total number of points for our player. We'll have a variable called points in the Game class:

```
// Keep track of the points for the player
float points = 0;
```

And we'll add to it whenever we hit an enemy:

```
if (collide != -1)
{
    enemyShipList.RemoveAt(i);
    playerFireballList.RemoveAt(collide);
    points += 200;
}
```

Here we just add 200 points every time an enemy is hit.

That sets us up for what we want to display for text on the screen, now to display it all. We'll keep the number of lives in the upper left corner of the screen and the total points in the upper right. So besides the Sprite Text object we'll need two position vectors. In the Game class add the following variables:

```
SpriteFont CourierNew;

// Location to draw the text
Vector2 textLeft;
Vector2 textMiddle;
```

CourierNew is the Sprite Font we created before and the other two are the text position vectors. To initialize them add the following to LoadContent():

```
 // Create the font
 CourierNew = Content.Load<SpriteFont>("ShipFont");

 // Set text positions
textMiddle = new Vector2(graphics.GraphicsDevice.Viewport.Width / 1.3f,
graphics.GraphicsDevice.Viewport.Height / 30);
textLeft = new Vector2(graphics.GraphicsDevice.Viewport.Width / 30,
 graphics.GraphicsDevice.Viewport.Height / 30);
```

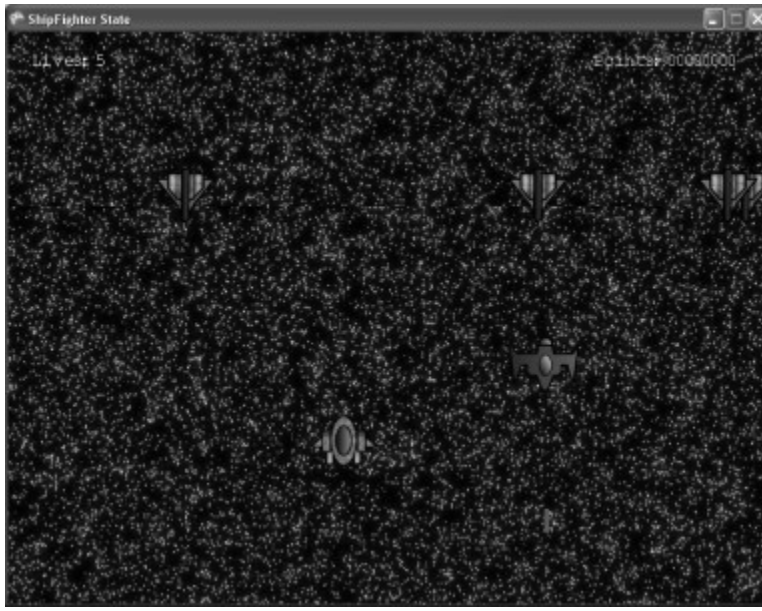
This just creates the font and the two position Vectors. The text position vectors use some rough calculations based on the window width to find their location. Next we'll add the following function to the Game class:

```
public void DrawText(SpriteBatch TextBatch)
{
    string output = "Lives: " + playerShip.Lives.ToString();

    TextBatch.DrawString(CourierNew, output, textLeft, Color.White);

    string pointString = points.ToString();

    for (int i = pointString.Length; i < 8; i++)
        pointString = "0" + pointString;

    pointString = "Points: " + pointString;

    TextBatch.DrawString(CourierNew, pointString, textMiddle, Color.White);
}
```

The main part of this is the SpriteBatch.DrawString() method. It works by calling the following:

```
spriteBatch.DrawString( SpriteFont, string textToPrint, Vector2 position, Text
    Color);
```

This method is simple enough, we just tell the SpriteBatch what font we want to use for the text, then give it a string of text to draw on the screen. We also tell it a position for the text and a color for it. In our ship game we use DrawText to draw the lives and points. The output looks like:



Adding Splash Screens

Now we have some HUD text up, let's add some splash screens. We'll add a start screen and an end screen, to make the game more complete for now.

But what we're really adding is more states to our game. If you recall the discussion from early in this section we can think of things in a game based on what state they are currently in. For the *Ship Fighter* game we'll create three possible states the game can be in, a start state: that occurs when you start the game, a running state that is when the game is running (so far all of our games have just been in the running state) and an end state, that occurs when the player runs out of lives or runs out of time (we'll put in a timeout feature.) The start state will just draw a big sprite over the entire screen and wait for a key press. The end state will also just draw another splash screen and wait for a key press to restart.

To start add the following enum to Game1.cs:

```
public enum GameState
{
	StartScreen,
	Running,
	EndScreen
}
```

This structure is called an enum. Enums work by creating a variable name and list of values. Then we can create a variable of the type and it can only have the values we specified. This is clearer if you look at the one we make for the game state; it helps us write code for the different states we can be in. In the game class we'll add a variable to track the current state:

```
GameState gameState = GameState.StartScreen;
```

The variable gameState can have one of three values, StartScreen, Running, or EndScreen. Let's go ahead and add the variables to hold the sprites for our splash screens and a float to track the entire time and the totalTime the game can run (endTime):

```
// total time tracker and endTime (timeout)
float endTime = 10.0f;

// Textures for our splash screen
Texture2D startScreen;
Texture2D endScreen;
```

For the two textures we'll need to load them in the LoadContent method:

```
startScreen = content.Load<Texture2D>("startGameSplash");
endScreen = content.Load<Texture2D>("endGameSplash");
```

For the different game states we'll choose between them by using ifs in the Draw and Update loop based on the current game state. For the Draw it will just be:

```
protected override void Draw(GameTime gameTime)
{
        graphics.GraphicsDevice.Clear(Color.CornflowerBlue);

        if (gameState == GameState.StartScreen)
        {
            DrawStartScreen();
        }
        else if (gameState == GameState.Running)
        {
            spaceBackground.Draw();

            spriteBatch.Begin();

            foreach (Fireball f in playerFireballList)
                f.Draw(spriteBatch);

            playerShip.Draw(spriteBatch);

            foreach (Fireball f in enemyFireballList)
                f.Draw(spriteBatch);

            foreach (Enemy e in enemyShipList)
                e.Draw(spriteBatch);

            DrawText(spriteBatch);

            spriteBatch.End();
        }
        else if (gameState == GameState.EndScreen)
        {
            DrawEndScreen();
        }
        base.Draw(gameTime);
    }
```

We just switch what we're doing based on the current state. If the current state is StartScreen we call

DrawStartScreen (a method we need to add). If we're in the running state we just draw as we were before. And if we're in the Endstate we draw the endscreen. The DrawStartScreen and DrawEndScreen methods are pretty simple, they just draw the appropriate sprite onscreen:

```
// Draw the start screen
private void DrawStartScreen()
{
    spriteBatch.Begin();

    spriteBatch.Draw(startScreen, Vector2.Zero, null, Color.White);

    spriteBatch.End();
}

// Draw the end screen
private void DrawEndScreen()
{
    spriteBatch.Begin();

    spriteBatch.Draw(endScreen, Vector2.Zero, null, Color.White);

    spriteBatch.End();
}
```

Likewise for the Update loop we'll toggle what we do based on the current state. After the time Update in the Update loop we add the following:

```
 // If we are starting or ending just update Splash screen
if (gameState == GameState.StartScreen || gameState ==
GameState.EndScreen)
      UpdateSplashScreen();
else // GameState is running
{
```

And the rest of this is just the regular Update loop. So if we are not in the running state we just call the update function as usual, but we'll add a line to check if the player lives are below zero, if they are we'll set the state to End. When the game is over we also want to reset our game to a start state. We'll put the player back in its start position and clear out all of the enemies and fireballs. Here is the code:

```
if (playerShip.Lives < 0 || totalTime > endTime)
{
    gameState = GameState.EndScreen;
    playerShip.Position = new Vector2( 400.0f, 500.0f );
```

```
    enemyFireballList.Clear();
    enemyShipList.Clear();
    playerFireballList.Clear();
}
```

For the UpdateSplashScreen method we'll see if the start button on the Xbox controller is hit or if the space bar is pressed on the keyboard. If either of those is true we just switch the current state to running.

```
private void UpdateSplashScreen()
{
    KeyboardState keyState = Keyboard.GetState();
    if (GamePad.GetState(PlayerIndex.One).Buttons.Start ==
            ButtonState.Pressed || keyState.IsKeyDown(Keys.Space))
        {
            gameState = GameState.Running;
            totalTime = 0.0f;
        }
}
```

Now if you run the game you get

We have a pretty complete game setup so far. Once last feature is a special effect. The special effect we'll add in is a particle explosion for whenever the enemy ships are hit. Next let's take a look at particle systems.

Particle Systems

Some effects we want to model in video games are made up of many small pieces, particles, interacting. Water and smoke (and explosions) are examples of this. To create explosions we could create an animation, drawing out every frame of the animation. The problem with that approach is that each explosion should be a little bit different. Another approach is to use particle systems. Particle systems can get very complex, but the basic concept is pretty simple. You create a list of particles, each one having a start state (color), end state, and velocity. Over time you update each particle individually and this creates an effect overall. For the explosion effect in our game here is a simple source file, BasicParticleSystem.cs, that does this for explosions:

```
class Particle
{
    Vector4 color;
    Vector4 startColor;
    Vector4 endColor;
    TimeSpan endTime = TimeSpan.Zero;
    TimeSpan lifetime;
    public Vector3 position;
    Vector3 velocity;
    protected Vector3 acceleration = new Vector3( 1.0f, 1.0f, 1.0f );

    public bool Delete;
    public Vector4 Color
    {
        get
        {
            return color;
        }
    }
```

```
    public Particle(Vector2 position2, Vector2 velocity2, Vector4 startColor,
        Vector4 endColor, TimeSpan lifetime)
    {
         velocity = new Vector3(velocity2, 0.0f);
         position = new Vector3(position2, 0.0f);
         this.startColor = startColor;
         this.endColor = endColor;
         this.lifetime = lifetime;
    }

    public void Update(TimeSpan time, TimeSpan elapsedTime)
    {
        //Start the animation 1st time round
        if (endTime == TimeSpan.Zero)
        {
            endTime = time + lifetime;
        }
        if (time > endTime)
        {
            Delete = true;
        }
        float percentLife = (float)((endTime.TotalSeconds - time.TotalSeconds)
            / lifetime.TotalSeconds);

        color = Vector4.Lerp(endColor, startColor, percentLife);

        velocity += Vector3.Multiply(acceleration,
            (float)elapsedTime.TotalSeconds);

        position += Vector3.Multiply(velocity,
            (float)elapsedTime.TotalSeconds);
      }
  }


class BasicParticleSystem
{
    static Random random = new Random();

    List<Particle> particleList = new List<Particle>();

    Texture2D circle;
    int Count = 0;

    public BasicParticleSystem(Texture2D circle)
    {
```

```
        this.circle = circle;
    }

    public void AddExplosion(Vector2 position)
    {
        for (int i = 0; i < 300; i++)
        {
            Vector2 velocity2 = (float)random.Next(100) *
                Vector2.Normalize(new Vector2((float)(random.NextDouble() –
                    .5), (float)(random.NextDouble() - .5)));

            particleList.Add(new Particle(
                    position,
                    velocity2,
                    (i > 70) ? new Vector4(1.0f, 0f, 0f, 1) : new Vector4(.941f,
                        .845f, 0f, 1),
                    new Vector4(.2f, .2f, .2f, 0f),
                    new TimeSpan(0, 0, 0, 0, random.Next(1000) + 500)));
             Count++;
        }
    }


public void Update(TimeSpan time, TimeSpan elapsed)
{
    if (Count > 0)
    {
        for( int i = 0; i < particleList.Count; i++ )
        {
            particleList[i].Update(time, elapsed);

            if (particleList[i].Delete) particleList.RemoveAt(i);
        }

        Count = particleList.Count;
        }
    }

    public void Draw(SpriteBatch batch)
    {
        if (Count != 0)
        {

            int particleCount = 0;

            foreach (Particle particle in particleList)
```

```
            {
                batch.Draw(circle,
                    new Vector2(particle.position.X, particle.position.Y),
                    null, new Color(((Particle)particle).Color), 0,
                    new Vector2(16, 16), .2f,
                    SpriteEffects.None, particle.position.Z);
                particleCount++;
            }
        }
    }
}
```

Again, we won't step through this code but looking at it quickly you can see that it has a particle class and a BasicParticleSystem class that let's us add an explosion and draw/update it. If you add this file to the ShipFighter project then adding it to the Game class is pretty straightforward. (Also add in the "circle.tga" art file to the project, as we'll use this for the picture for our particles.) First in the Game class add the following variable:

```
BasicParticleSystem particleSystem;
TimeSpan totalTimeSpan = new TimeSpan();
```

This declares our particle system. We are also going to be using a time variable to track the overall time for updating our system, so we added the totalTimeSpan structure. The next step is to initialize the particle system. In the LoadContent method of the game class add in:

```
particleSystem = new
BasicParticleSystem(Content.Load<Texture2D>("circle"));
```

This creates our particle system. Then we need to draw and update it. In the draw loop where we are using our spaceSpriteBatch add the following:

```
particleSystem.Draw(spriteBatch);
```

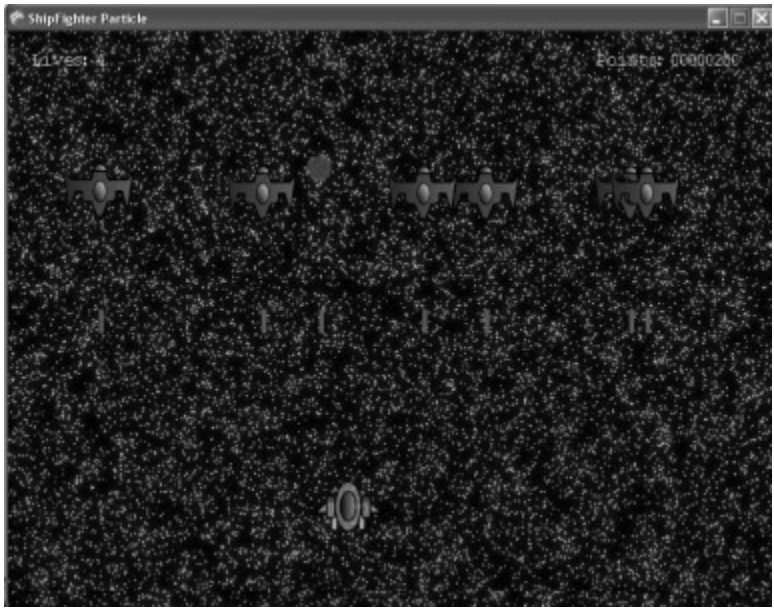
And in the update loop add:

```
totalTimeSpan += gameTime.ElapsedGameTime;
particleSystem.Update(totalTimeSpan, gameTime.ElapsedGameTime);
```

This updates our time for the system and the particle system itself. The last thing to do is to add an explosion to the system whenever an enemy is hit by a fireball or crashes into the player. Here is a modification of the collision detection part of the update loop that does this:

```
int collide = enemyShipList[i].CollisionBall(playerFireballList);

if (collide != -1)
{
    particleSystem.AddExplosion(enemyShipList[i].Position);

    enemyShipList.RemoveAt(i);
    playerFireballList.RemoveAt(collide);
    points += 200;
}
else
if (playerShip.CollisionTest(enemyShipList[i].Position,
enemyShipList[i].Radius))
{
    particleSystem.AddExplosion(enemyShipList[i].Position);
    enemyShipList.RemoveAt(i);
}
else if (enemyShipList[i].Position.Y > 2000.0f)
    enemyShipList.RemoveAt(i);
```

This sets up our explosions. If you run the game now you'll see them whenever you hit an enemy:



End

This brings us to the end of the game *Ship Fighter*. We've only scratched the surface of game programming. Good luck in future game programming.